

Alexandru Vaduva, Alex González,
Chris Simmonds

Linux: Embedded Development

Learning Path

Leverage the power of Linux to develop captivating and powerful embedded Linux projects



Packt

Linux: Embedded Development

Leverage the power of Linux to develop captivating and powerful embedded Linux projects

A course in three modules

Packt

BIRMINGHAM - MUMBAI

Linux: Embedded Development

Copyright © 2016 Packt Publishing

All rights reserved. No part of this course may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this course to ensure the accuracy of the information presented. However, the information contained in this course is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this course.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this course by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Published on: September 2016

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78712-420-2

www.packtpub.com

Credits

Authors

Alexandru Vaduva
Alex González
Chris Simmonds

Content Development Editor

Rohit Kumar Singh

Graphics

Jason Monteiro

Reviewers

Peter Ducai
Alex Tereschenko
Burt Janz
Dave (Jing) Tian
Javier Viguera
Robert Berger
Tim Bird
Mathieu Deschamps
Mark Furman
Klaas van Gend
Behan Webster

Production Coordinator

Aparna Bhagat

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.

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.

The Linux kernel is at the heart of a large number of embedded products being designed today. Over the last 10 years, this operating system has developed from dominating the server market to being the most used operating system in embedded systems, even those with real-time requirements.

But at the same time, an embedded Linux product is not only the Linux kernel. Companies need to build an embedded system over the operating system, and that's where embedded Linux was finding it difficult to make its place – until Yocto arrived.

The Yocto Project brings all the benefits of Linux into the development of embedded systems. It provides a standard build system that allows you to develop embedded products in a quick, reliable, and controlled way.

What this learning path covers

Module 1, Learning Embedded Linux Using the Yocto, introduces you to embedded Linux software and hardware architecture, cross compiling, bootloader. You will also get an overview of the available Yocto Project components.

Module 2, Embedded Linux Projects Using Yocto Project, helps you set up and configure the Yocto Project tools. You will learn the methods to share source code and modifications

Module 3, Mastering Embedded Linux, takes you through the product cycle and gives you an in-depth description of the components and options that are available at each stage.

What you need for this learning path

Before reading this learning path, prior knowledge of embedded Linux and Yocto would be helpful, though not mandatory. In this learning path, a number of exercises are available, and to do them, a basic understanding of the GNU/Linux environment would be useful.

The examples have been tested with an Ubuntu 14.04 LTS system, but any Linux distribution supported by the Yocto Project can be used. Any piece of i.MX-based hardware can be used to follow the examples.

The versions of the main packages for the target are U-Boot 2015.07, Linux 4.1, Yocto Project 1.8 "Fido", and Buildroot 2015.08.

Who this learning path is for

If you are a developer who wants to build embedded systems using Linux, this learning path is for you. It is an ideal guide for you to become proficient and broaden your knowledge with examples that are immediately applicable to your embedded developments. A basic understanding of C programming and experience with systems programming is needed. Experienced embedded Yocto developers will find new insight into working methodologies and ARM specific development competence.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this learning path – what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt product, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this learning path from your account at <http://www.packtpub.com>. If you purchased this learning path elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the learning path in the **Search** box.
5. Select the learning path for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this learning path from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the learning path's webpage at the Packt Publishing website. This page can be accessed by entering the learning path's name in the **Search** box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the learning path is also hosted on GitHub at <https://github.com/PacktPublishing/Embedded-Linux-for-Developers>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our learning path – maybe a mistake in the text or the code – we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this learning path. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your learning path, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the learning path in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this learning path, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

Module 1: Learning Embedded Linux Using the Yocto Project

Chapter 1: Introduction	3
Advantages of Linux and open source systems	3
Embedded systems	5
Introducing GNU/Linux	8
Introduction to the Yocto Project	11
Summary	20
Chapter 2: Cross-compiling	21
Introducing toolchains	21
Components of toolchains	23
Delving into C libraries	31
Working with toolchains	34
The Yocto Project reference	40
Summary	45
Chapter 3: Bootloaders	47
The role of the bootloader	48
Comparing various bootloaders	50
Delving into the bootloader cycle	51
The U-Boot bootloader	53
The Yocto Project	63
Summary	64

Chapter 4: Linux Kernel	65
The role of the Linux kernel	68
Delving into the features of the Linux kernel	69
Interrupts	83
Timers	88
Linux kernel interaction	89
Kernel sources	94
Devices and modules	97
Debugging a kernel	101
The Yocto Project reference	102
Summary	106
Chapter 5: The Linux Root Filesystem	107
Interacting with the root filesystem	107
Understanding BusyBox	123
Minimal root filesystem	125
The Yocto Project	127
Summary	129
Chapter 6: Components of the Yocto Project	131
Poky	131
Eclipse ADT plug-ins	136
Hob and Toaster	140
Autobuilder	141
Lava	141
Wic	142
Summary	143
Chapter 7: ADT Eclipse Plug-ins	145
The Application Development Toolkit	146
Eclipse IDE	154
Summary	173
Chapter 8: Hob, Toaster, and Autobuilder	175
Hob	175
Toaster	188
Autobuilder	195
Summary	197
Chapter 9: Wic and Other Tools	199
Swabber	200
Wic	204
LAVA	209
Summary	212

Table of Contents

Chapter 10: Real-time	213
Understanding GPOS and RTOS	214
PREEMPT_RT	217
Linux real-time applications	229
Benchmarking	230
Meta-realtime	231
Summary	233
Chapter 11: Security	235
Security in Linux	236
SELinux	236
Grsecurity	239
Security for the Yocto Project	246
Meta-security and meta-selinux	247
Summary	256
Chapter 12: Virtualization	257
Linux virtualization	258
Virtualization support for the Yocto Project	266
Summary	282
Chapter 13: CGL and LSB	283
Linux Standard Base	284
Carrier grade options	290
Specific support for the Yocto Project	295
Summary	304

Module 2: Embedded Linux Projects Using Yocto Project Cookbook

Chapter 1: The Build System	307
Introduction	308
Setting up the host system	309
Installing Poky	311
Creating a build directory	313
Building your first image	315
Explaining the Freescale Yocto ecosystem	318
Installing support for Freescale hardware	321
Building Wandboard images	323
Troubleshooting your Wandboard's first boot	325
Configuring network booting for a development setup	327

Table of Contents

Sharing downloads	329
Sharing the shared state cache	330
Setting up a package feed	332
Using build history	335
Working with build statistics	337
Debugging the build system	340
Chapter 2: The BSP Layer	347
Introduction	348
Creating a custom BSP layer	348
Introducing system development workflows	353
Adding a custom kernel and bootloader	356
Explaining Yocto's Linux kernel support	360
Describing Linux's build system	363
Configuring the Linux kernel	364
Building the Linux kernel	369
Building external kernel modules	373
Debugging the Linux kernel and modules	377
Debugging the Linux kernel booting process	381
Using the kernel function tracing system	385
Managing the device tree	390
Debugging device tree issues	397
Chapter 3: The Software Layer	401
Introduction	401
Exploring an image's contents	402
Adding a new software layer	403
Selecting a specific package version and providers	407
Adding supported packages	409
Adding new packages	412
Adding data, scripts, or configuration files	418
Managing users and groups	419
Using the sysvinit initialization manager	423
Using the systemd initialization manager	425
Installing package-installation scripts	428
Reducing the Linux kernel image size	429
Reducing the root filesystem image size	433
Releasing software	436
Analyzing your system for compliance	439
Working with open source and proprietary code	440

Chapter 4: Application Development	445
Introduction	446
Preparing and using an SDK	446
Using the Application Development Toolkit	448
Using the Eclipse IDE	450
Developing GTK+ applications	455
Using the Qt Creator IDE	462
Developing Qt applications	466
Describing workflows for application development	472
Working with GNU make	477
Working with the GNU build system	478
Working with the CMake build system	480
Working with the SCons builder	481
Developing with libraries	482
Working with the Linux framebuffer	489
Using the X Windows system	495
Using Wayland	498
Adding Python applications	501
Integrating the Oracle Java Runtime Environment	505
Integrating the Open Java Development Kit	509
Integrating Java applications	511
Chapter 5: Debugging, Tracing, and Profiling	517
Introduction	518
Analyzing core dumps	518
Native GDB debugging	522
Cross GDB debugging	525
Using strace for application debugging	530
Using the kernel's performance counters	531
Using static kernel tracing	535
Using dynamic kernel tracing	542
Using dynamic kernel events	551
Exploring Yocto's tracing and profiling tools	555
Tracing and profiling with perf	557
Using SystemTap	568
Using OProfile	571
Using LTTng	576
Using blktrace	583

Module 3: Mastering Embedded Linux Programming

Chapter 1: Starting Out	591
Selecting the right operating system	593
The players	593
Project lifecycle	595
Open source	596
Hardware for embedded Linux	598
Hardware used in this book	599
Software used in this book	602
Summary	602
Chapter 2: Learning About Toolchains	603
What is a toolchain?	603
Types of toolchain - native versus cross toolchain	605
Choosing the C library	608
Finding a toolchain	610
Anatomy of a toolchain	614
Other tools in the toolchain	617
Looking at the components of the C library	618
Linking with libraries: static and dynamic linking	619
The art of cross compiling	623
Problems with cross compiling	628
Summary	629
Chapter 3: All About Bootloaders	631
What does a bootloader do?	631
The boot sequence	632
Booting with UEFI firmware	636
Moving from bootloader to kernel	636
Introducing device trees	637
Choosing a bootloader	643
U-Boot	643
Barebox	656
Summary	658
Chapter 4: Porting and Configuring the Kernel	659
What does the kernel do?	659
Choosing a kernel	661
Building the kernel	664
Compiling	671
Cleaning kernel sources	675

Booting your kernel	676
Porting Linux to a new board	680
Additional reading	684
Summary	684
Chapter 5: Building a Root Filesystem	685
What should be in the root filesystem?	686
Programs for the root filesystem	691
Libraries for the root filesystem	695
Device nodes	697
The proc and sysfs filesystems	698
Kernel modules	699
Transferring the root filesystem to the target	700
Creating a boot ramdisk	700
The init program	705
Configuring user accounts	706
Starting a daemon process	708
A better way of managing device nodes	708
Configuring the network	710
Creating filesystem images with device tables	712
Mounting the root filesystem using NFS	714
Using TFTP to load the kernel	716
Additional reading	717
Summary	718
Chapter 6: Selecting a Build System	719
No more rolling your own embedded Linux	719
Build systems	720
Package formats and package managers	722
Buildroot	722
The Yocto Project	731
Further reading	746
Summary	747
Chapter 7: Creating a Storage Strategy	749
Storage options	749
Accessing flash memory from the bootloader	755
Accessing flash memory from Linux	756
Filesystems for flash memory	763
Filesystems for NOR and NAND flash memory	764
Filesystems for managed flash	773
Read-only compressed filesystems	778
Temporary filesystems	779

Table of Contents

Making the root filesystem read-only	780
Filesystem choices	781
Updating in the field	782
Further reading	785
Summary	786
<u>Chapter 8: Introducing Device Drivers</u>	<u>787</u>
The role of device drivers	787
Character devices	788
Block devices	790
Network devices	792
Finding out about drivers at runtime	793
Finding the right device driver	798
Device drivers in user-space	799
Writing a kernel device driver	806
Loading kernel modules	812
Discovering hardware configuration	812
Additional reading	817
Summary	817
<u>Chapter 9: Starting up - the init Program</u>	<u>819</u>
After the kernel has booted	819
Introducing the init programs	820
BusyBox init	821
System V init	823
systemd	829
Further reading	836
Summary	836
<u>Chapter 10: Learning About Processes and Threads</u>	<u>837</u>
Process or thread?	837
Processes	839
Threads	850
Scheduling	857
Further reading	861
Summary	862
<u>Chapter 11: Managing Memory</u>	<u>863</u>
Virtual memory basics	863
Kernel space memory layout	864
User space memory layout	868
Process memory map	869
Swap	871

Mapping memory with mmap	872
How much memory does my application use?	874
Per-process memory usage	875
Identifying memory leaks	878
Running out of memory	881
Further reading	883
Summary	883
Chapter 12: Debugging with GDB	885
The GNU debugger	885
Preparing to debug	886
Debugging applications using GDB	887
Remote debugging using gdbserver	887
Starting to debug	889
Debugging shared libraries	893
Just-in-time debugging	895
Debugging forks and threads	895
Core files	896
GDB user interfaces	898
Debugging kernel code	901
Additional reading	910
Summary	911
Chapter 13: Profiling and Tracing	913
The observer effect	913
Beginning to profile	915
Profiling with top	915
Introducing perf	917
Other profilers: OProfile and gprof	924
Tracing events	926
Introducing Ftrace	927
Using LTTng	933
Using Valgrind for application profiling	937
Callgrind	937
Helgrind	938
Using strace to show system calls	939
Summary	941
Chapter 14: Real-time Programming	943
What is real-time?	943
Identifying the sources of non-determinism	946
Understanding scheduling latency	947
Kernel preemption	948

Table of Contents —————

The real-time Linux kernel (PREEMPT_RT)	949
Threaded interrupt handlers	950
Preemptible kernel locks	952
Getting the PREEMPT_RT patches	952
High resolution timers	954
Avoiding page faults in a real-time application	954
Interrupt shielding	956
Measuring scheduling latencies	956
Further reading	962
Summary	963
Bibliography	965



Module 1

Learning Embedded Linux Using the Yocto Project

Develop powerful embedded Linux systems with the Yocto Project components

1

Introduction

In this chapter, you will be presented with the advantages of Linux and open source development. There will be examples of systems running embedded Linux, which a vast number of embedded hardware platforms support. After this, you will be introduced to the architecture and development environment of an embedded Linux system, and, in the end, the Yocto Project, where its Poky build system's properties and purposes are summarized.

Advantages of Linux and open source systems

Most of the information available in this book, and the examples presented as exercises, have one thing in common: the fact that they are freely available for anyone to access. This book tries to offer guidance to you on how to interact with existing and freely available packages that could help an embedded engineer, such as you, and at the same time, also try to arouse your curiosity to learn more.



More information on open source can be gathered from the **Open Source Initiative (OSI)** at <http://opensource.org/>.

The main advantage of open source is represented by the fact that it permits developers to concentrate more on their products and their added value. Having an open source product offers access to a variety of new possibilities and opportunities, such as reduced costs of licensing, increased skills, and knowledge of a company. The fact that a company uses an open source product that most people have access to, and can understand its working, implies budget savings. The money saved could be used in other departments, such as hardware or acquisitions.

Usually, there is a misconception about open source having little or no control over a product. However, the opposite is true. The open source system, in general, offers full control over software, and we are going to demonstrate this. For any software, your open source project resides on a repository that offers access for everyone to see. Since you're the person in charge of a project, and its administrator as well, you have all the right in the world to accept the contributions of others, which lends them the same right as you, and this basically gives you the freedom to do whatever you like. Of course, there could be someone who is inspired by your project and could do something that is more appreciated by the open source community. However, this is how progress is made, and, to be completely honest, if you are a company, this kind of scenario is almost invalid. Even in this case, this situation does not mean the death of your project, but an opportunity instead. Here, I would like to present the following quote:

"If you want to build an open source project, you can't let your ego stand in the way. You can't rewrite everybody's patches, you can't second-guess everybody, and you have to give people equal control."

– Rasmus Lerdorf

Allowing access to others, having external help, modifications, debugging, and optimizations performed on your open source software implies a longer life for the product and better quality achieved over time. At the same time, the open source environment offers access to a variety of components that could easily be integrated in your product if there's a requirement for them. This permits a quick development process, lower costs, and also shifts a great deal of the maintenance and development work from your product. Also, it offers the possibility to support a particular component to make sure that it continues to suit your needs. However, in most instances, you would need to take some time and build this component for your product from zero.

This brings us to the next benefit of open source, which involves testing and quality assurance for our product. Besides the lesser amount of work that is needed for testing, it is also possible to choose from a number of options before deciding which components fits best for our product. Also, it is cheaper to use open source software, than buying and evaluating proprietary products. This takes and gives back process, visible in the open source community, is the one that generates products of a higher quality and more mature ones. This quality is even greater than that of other proprietary or closed source similar products. Of course, this is not a generally valid affirmation and only happens for mature and widely used products, but here appears the term community and foundation into play.

In general, open source software is developed with the help of communities of developers and users. This system offers access to a greater support on interaction with the tools directly from developers - the sort of thing that does not happen when working with closed source tools. Also, there is no restriction when you're looking for an answer to your questions, no matter whether you work for a company or not. Being part of the open source community means more than bug fixing, bug reporting, or feature development. It is about the contribution added by the developers, but, at the same time, it offers the possibility for engineers to get recognition outside their working environment, by facing new challenges and trying out new things. It can also be seen as a great motivational factor and a source of inspiration for everyone involved in the process.

Instead of a conclusion, I would also like to present a quote from the person who forms the core of this process, the man who offered us Linux and kept it open source:

"I think, fundamentally, open source does tend to be more stable software. It's the right way to do things."

- Linus Torvalds

Embedded systems

Now that the benefits of open source have been introduced to you, I believe we can go through a number of examples of embedded systems, hardware, software, and their components. For starters, embedded devices are available anywhere around us: take a look at your smartphone, car infotainment system, microwave oven, or even your MP3 player. Of course, not all of them qualify to be Linux operating systems, but they all have embedded components that make it possible for them to fulfill their designed functions.

General description

For Linux to be run on any device hardware, you will require some hardware-dependent components that are able to abstract the work for hardware-independent ones. The boot loader, kernel, and toolchain contain hardware-dependent components that make the performance of work easier for all the other components. For example, a BusyBox developer will only concentrate on developing the required functionalities for his application, and will not concentrate on hardware compatibility. All these hardware-dependent components offer support for a large variety of hardware architectures for both 32 and 64 bits. For example, the U-Boot implementation is the easiest to take as an example when it comes to source code inspection. From this, we can easily visualize how support for a new device can be added.

We will now try to do some of the little exercises presented previously, but before moving further, I must present the computer configuration on which I will continue to do the exercises, to make sure that you face as few problems as possible. I am working on an Ubuntu 14.04 and have downloaded the 64-bit image available on the Ubuntu website at <http://www.ubuntu.com/download/desktop>

Information relevant to the Linux operation running on your computer can be gathered using this command:

```
uname -srmpio
```

The preceding command generates this output:

```
Linux 3.13.0-36-generic x86_64 x86_64 x86_64 GNU/Linux
```

The next command to gather the information relevant to the Linux operation is as follows:

```
cat /etc/lsb-release
```

The preceding command generates this output:

```
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=14.04
DISTRIB_CODENAME=trusty
DISTRIB_DESCRIPTION="Ubuntu 14.04.1 LTS"
```

Examples

Now, moving on to exercises, the first one requires you fetch the git repository sources for the U-Boot package:

```
sudo apt-get install git-core
git clone http://git.denx.de/u-boot.git
```

After the sources are available on your machine, you can try to take a look inside the board directory; here, a number of development board manufacturers will be present. Let's take a look at board/atmel/sama5d3_xplained, board/faraday/a320evo, board/freescale/imx, and board/freescale/b4860qds. By observing each of these directories, a pattern can be visualized. Almost all of the boards contain a Kconfig file, inspired mainly from kernel sources because they present the configuration dependencies in a clearer manner. A maintainers file offers a list with the contributors to a particular board support. The base Makefile file takes from the higher-level makefiles the necessary object files, which are obtained after a board-specific support is built. The difference is with board/freescale/imx which only offers a list of configuration data that will be later used by the high-level makefiles.

At the kernel level, the hardware-dependent support is added inside the `arch` file. Here, for each specific architecture besides `Makefile` and `Kconfig`, various numbers of subdirectories could also be added. These offer support for different aspects of a kernel, such as the boot, kernel, memory management, or specific applications.

By cloning the kernel sources, the preceding information can be easily visualized by using this code:

```
git clone https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.  
git
```

Some of the directories that can be visualized are `arch/arc` and `arch/metag`.

From the toolchain point of view, the hardware-dependent component is represented by the GNU C Library, which is, in turn, usually represented by `glibc`. This provides the system call interface that connects to the kernel architecture-dependent code and further provides the communication mechanism between these two entities to user applications. System calls are presented inside the `sysdeps` directory of the `glibc` sources if the `glibc` sources are cloned, as follows:

```
git clone http://sourceware.org/git/glibc.git
```

The preceding information can be verified using two methods: the first one involves opening the `sysdeps/arm` directory, for example, or by reading the `ChangeLog.old-ports-arm` library. Although it's old and has nonexistent links, such as `ports` directory, which disappeared from the newer versions of the repository, the latter can still be used as a reference point.

These packages are also very easily accessible using the Yocto Project's `poky` repository. As mentioned at <https://www.yoctoproject.org/about>:

"The Yocto Project is an open source collaboration project that provides templates, tools and methods to help you create custom Linux-based systems for embedded products regardless of the hardware architecture. It was founded in 2010 as a collaboration among many hardware manufacturers, open-source operating systems vendors, and electronics companies to bring some order to the chaos of embedded Linux development."

Most of the interaction anyone has with the Yocto Project is done through the `Poky` build system, which is one of its core components that offers the features and functionalities needed to generate fully customizable Linux software stacks. The first step needed to ensure interaction with the repository sources would be to clone them:

```
git clone -b dizzy http://git.yoctoproject.org/git/poky
```

After the sources are present on your computer, a set of recipes and configuration files need to be inspected. The first location that can be inspected is the U-Boot recipe, available at `meta/recipes-bsp/u-boot/u-boot_2013.07.bb`. It contains the instructions necessary to build the U-Boot package for the corresponding selected machine. The next place to inspect is in the recipes available in the kernel. Here, the work is sparse and more package versions are available. It also provides some bbappends for available recipes, such as `meta/recipes-kernel/linux/linux-yocto_3.14.bb` and `meta-yocto-bsp/recipes-kernel/linux/linux-yocto_3.10.bbappend`. This constitutes a good example for one of the kernel package versions available when starting a new build using BitBake.

Toolchain construction is a big and important step for host generated packages. To do this, a set of packages are necessary, such as `gcc`, `binutils`, `glibc` library, and `kernel` headers, which play an important role. The recipes corresponding to this package are available inside the `meta/recipes-devtools/gcc/`, `meta/recipes-devtools/binutils`, and `meta/recipes-core/glibc` paths. In all the available locations, a multitude of recipes can be found, each one with a specific purpose. This information will be detailed in the next chapter.

The configurations and options for the selection of one package version in favor of another is mainly added inside the machine configuration. One such example is the Freescale MPC8315E-rdb low-power model supported by Yocto 1.6, and its machine configuration is available inside the `meta-yocto-bsp/conf/machine/mpc8315e-rdb.conf` file.



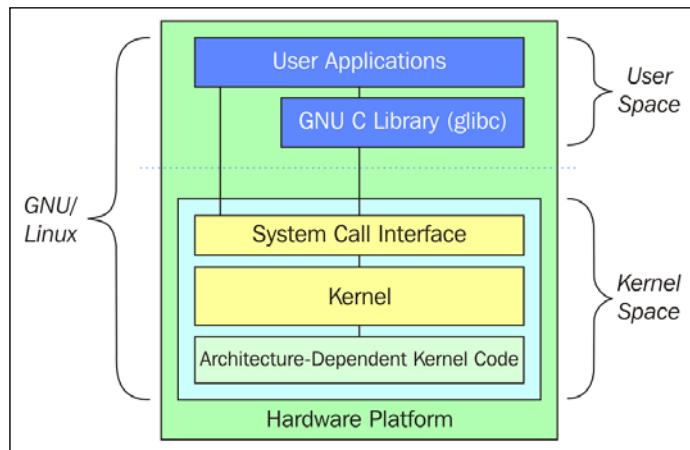
More information on this development board can be found at http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MPC8315E.

Introducing GNU/Linux

GNU/Linux, or Linux as it's commonly known, represents a name that has a long line of tradition behind it, and is one of the most important unions of open source software. Shortly, you will be introduced to the history of what is offered to people around the world today and the choice available in terms of selecting personal computer operating systems. Most of all, we will look at what is offered to hardware developers and the common ground available for the development of platforms.

GNU/Linux consists of the Linux kernel and has a collection of user space applications that are put on top of GNU C Library; this acts as a computer operating system. It may be considered as one of the most prolific instances of open source and free software available, which is still in development. Its history started in 1983 when Richard Stallman founded the GNU Project with the goal of developing a complete Unix-like operating system, which could be put together only from free software. By the beginning of the 1990s, GNU already offered a collection of libraries, Unix-like shells, compilers, and text editors. However, it lacked a kernel. They started developing their own kernel, the Hurd, in 1990. The kernel was based on a Mach micro-kernel design, but it proved to be difficult to work with and had a slow development process.

Meanwhile, in 1991, a Finnish student started working on another kernel as a hobby while attending the University of Helsinki. He also got help from various programmers who contributed to the cause over the Internet. That student's name was Linus Torvalds and, in 1992, his kernel was combined with the GNU system. The result was a fully functional operating system called GNU/Linux that was free and open source. The most common form of the GNU system is usually referred to as a *GNU/Linux system*, or even a *Linux distribution*, and is the most popular variant of GNU. Today, there are a great number of distributions based on GNU and the Linux kernel, and the most widely used ones are: Debian, Ubuntu, Red Hat Linux, SuSE, Gentoo, Mandriva, and Slackware. This image shows us how the two components of Linux work together:



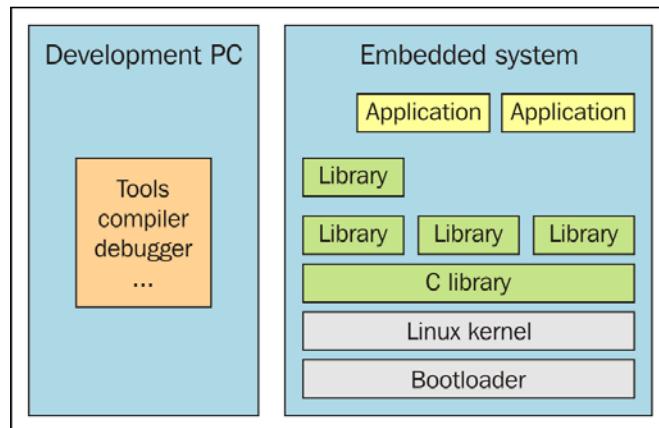
Introduction

Although not originally envisioned to run on anything else than x86 PCs, today, the Linux operating system is the most widespread and portable operating system. It can be found on both embedded devices or supercomputers because it offers freedom to its users and developers. Having tools to generate customizable Linux systems is another huge step forward in the development of this tool. It offers access to the GNU/Linux ecosystem to new categories of people who, by using a tool, such as BitBake, end up learning more about Linux, its architecture differences, root filesystem construction and configuration, toolchains, and many other things present in the Linux world.

Linux is not designed to work on microcontrollers. It will not work properly if it has less than 32 MB of RAM, and it will need to have at least 4 MB of storage space. However, if you take a look at this requirement, you will notice that it is very permissive. Adding to this is the fact that it also offers support for a variety of communication peripherals and hardware platforms, which gives you a clear image of why it is so widely adopted.

[ Well, it may work on 8MB of RAM, but that depends on the application's size as well.]

Working with a Linux architecture in an embedded environment requires certain standards. This is an image that represents graphically an environment which was made available on one of free-electrons Linux courses:



The preceding image presents the two main components that are involved in the development process when working with Linux in the embedded devices world:

- **Host machine:** This is the machine where all the development tools reside. Outside the Yocto world, these tools are represented by a corresponding toolchain cross-compiled for a specific target and its necessary applications sources and patches. However, for an Yocto user, all these packages, and the preparation work involved, is reduced to automatized tasks executed before the actual work is performed. This, of course, has to be prioritized adequately.
- **Target machine:** This is the embedded system on which the work is done and tested. All the software available on the target is usually cross-compiled on the host machine, which is a more powerful and more efficient environment. The components that are usually necessary for an embedded device to boot Linux and operate various application, involve using a bootloader for basic initiation and loading of the Linux kernel. This, in turn, initializes drivers and the memory, and offers services for applications to interact with through the functions of the available C libraries.



There are also other methods of working with embedded devices, such as cross-canadian and native development, but the ones presented here are the most used and offer the best results for both developers and companies when it comes to software development for embedded devices.

To have a functional Linux operating system on an development board, a developer first needs to make sure that the kernel, bootloader, and board corresponding drives are working properly before starting to develop and integrate other applications and libraries.

Introduction to the Yocto Project

In the previous section, the benefits of having an open source environment were presented. Taking a look at how embedded development was done before the advent of the Yocto Project offers a complete picture of the benefits of this project. It also gives an answer as to why it was adopted so quickly and in such huge numbers.

Using the Yocto Project, the whole process gets a bit more automatic, mostly because the workflow permitted this. Doing things manually requires a number of steps to be taken by developers:

1. Select and download the necessary packages and components.
2. Configure the downloaded packages.
3. Compile the configured packages.
4. Install the generated binary, libraries, and so on, on `rootfs` available on development machine.
5. Generate the final deployable format.

All these steps tend to become more complex with the increase in the number of software packages that need to be introduced in the final deployable state. Taking this into consideration, it can clearly be stated that manual work is suitable only for a small number of components; automation tools are usually preferred for large and complex systems.

In the last ten years, a number of automation tools could be used to generate an embedded Linux distribution. All of them were based on the same strategy as the one described previously, but they also needed some extra information to solve dependency related problems. These tools are all built around an engine for the execution of tasks and contain metadata that describes actions, dependencies, exceptions, and rules.

The most notable solutions are Buildroot, Linux Target Image Builder (LTIB), Scratchbox, OpenEmbedded, Yocto, and Angstrom. However, Scratchbox doesn't seem to be active anymore, with the last commit being done in April 2012. LTIB was the preferred build tool for Freescale and it has lately moved more toward Yocto; in a short span of time, LTIB may become deprecated also.

Buildroot

Buildroot as a tool tries to simplify the ways in which a Linux system is generated using a cross-compiler. Buildroot is able to generate a bootloader, kernel image, root filesystem, and even a cross-compiler. It can generate each one of these components, although in an independent way, and because of this, its main usage has been restricted to a cross-compiled toolchain that generates a corresponding and custom root filesystem. It is mainly used in embedded devices and very rarely for x86 architectures; its main focus being architectures, such as ARM, PowerPC, or MIPS. As with every tool presented in this book, it is designed to run on Linux, and certain packages are expected to be present on the host system for their proper usage. There are a couple of mandatory packages and some optional ones as well.

There is a list of mandatory packages that contain the certain packages, and are described inside the Buildroot manual available at <http://buildroot.org/downloads/manual/manual.html>. These packages are as follows:

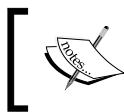
- `which`
- `sed`
- `make` (version 3.81 or any later ones)
- `binutils`
- `build-essential` (required for Debian-based systems only)
- `gcc` (version 2.95 or any later ones)
- `g++` (version 2.95 or any later ones)
- `bash`
- `patch`
- `gzip`
- `bzip2`
- `perl`(version 5.8.7 or any later ones)
- `tar`
- `cpio`
- `python`(version 2.6 or 2.7 ones)
- `unzip`
- `rsync`
- `wget`

Beside these mandatory packages, there are also a number of optional packages. They are very useful for the following:

- **Source fetching tools:** In an official tree, most of the package retrieval is done using `wget` from `http`, `https`, or even `ftp` links, but there are also a couple of links that need a version control system or another type of tool. To make sure that the user does not have a limitation to fetch a package, these tools can be used:
 - `bazaar`
 - `cvs`
 - `git`
 - `mercurial`
 - `rsync`

- scp
- subversion
- **Interface configuration dependencies:** They are represented by the packages that are needed to ensure that the tasks, such as kernel, BusyBox, and U-Boot configuration, are executed without problems:
 - ncurses5 is used for the menuconfig interface
 - qt4 is used for the xconfig interface
 - glib2, gtk2, and glade2 are used for the gconfig interface
- **Java related package interaction:** This is used to make sure that when a user wants to interact with the Java Classpath component, that it will be done without any hiccups:
 - javac: this refers to the Java compiler
 - jar: This refers to the Java archive tool
- **Graph generation tools:** The following are the graph generation tools:
 - graphviz to use graph-depends and <pkg>-graph-depends
 - python-matplotlib to use graph-build
- **Documentation generation tools:** The following are the tools necessary for the documentation generation process:
 - asciidoc, version 8.6.3 or higher
 - w3m
 - python with the argparse module (which is automatically available in 2.7+ and 3.2+ versions)
 - dblatex (necessary for pdf manual generation only)

Buildroot releases are made available to the open source community at <http://buildroot.org/downloads/> every three months, specifically in February, May, August, and November, and the release name has the buildroot-yyyy-mm format. For people interested in giving Buildroot a try, the manual described in the previous section should be the starting point for installing and configuration. Developers interested in taking a look at the Buildroot source code can refer to <http://git.buildroot.net/buildroot/>.



Before cloning the Buildroot source code, I suggest taking a quick look at <http://buildroot.org/download>. It could help out anyone who works with a proxy server.

Next, there will be presented a new set of tools that brought their contribution to this field and place on a lower support level the Buildroot project. I believe that a quick review of the strengths and weaknesses of these tools would be required. We will start with Scratchbox and, taking into consideration that it is already deprecated, there is not much to say about it; it's being mentioned purely for historical reasons. Next on the line is LTIB, which constituted the standard for Freescale hardware until the adoption of Yocto. It is well supported by Freescale in terms of **Board Support Packages (BSPs)** and contains a large database of components. On the other hand, it is quite old and it was switched with Yocto. It does not contain the support of new distributions, it is not used by many hardware providers, and, in a short period of time, it could very well become as deprecated as Scratchbox. Buildroot is the last of them and is easy to use, having a `Makefile` base format and an active community behind it. However, it is limited to smaller and simpler images or devices, and it is not aware of partial builds or packages.

OpenEmbedded

The next tools to be introduced are very closely related and, in fact, have the same inspiration and common ancestor, the OpenEmbedded project. All three projects are linked by the common engine called Bitbake and are inspired by the Gentoo Portage build tool. OpenEmbedded was first developed in 2001 when the Sharp Corporation launched the ARM-based PDA, and SL-5000 Zaurus, which run Lineo, an embedded Linux distribution. After the introduction of Sharp Zaurus, it did not take long for Chris Larson to initiate the OpenZaurus Project, which was meant to be a replacement for SharpROM, based on Buildroot. After this, people started to contribute many more software packages, and even the support of new devices, and, eventually, the system started to show its limitations. In 2003, discussions were initiated around a new build system that could offer a generic build environment and incorporate the usage models requested by the open source community; this was the system to be used for embedded Linux distributions. These discussions started showing results in 2003, and what has emerged today is the Openembedded project. It had packages ported from OpenZaurus by people, such as Chris Larson, Michael Lauer, and Holger Schurig, according to the capabilities of the new build system.

The Yocto Project is the next evolutionary stage of the same project and has the Poky build system as its core piece, which was created by Richard Purdie. The project started as a stabilized branch of the OpenEmbedded project and only included a subset of the numerous recipes available on OpenEmbedded; it also had a limited set of devices and support of architectures. Over time, it became much more than this: it changed into a software development platform that incorporated a fakeroot replacement, an Eclipse plug-in, and QEMU-based images. Both the Yocto Project and OpenEmbedded now coordinate around a core set of metadata called **OpenEmbedded-Core (OE-Core)**.

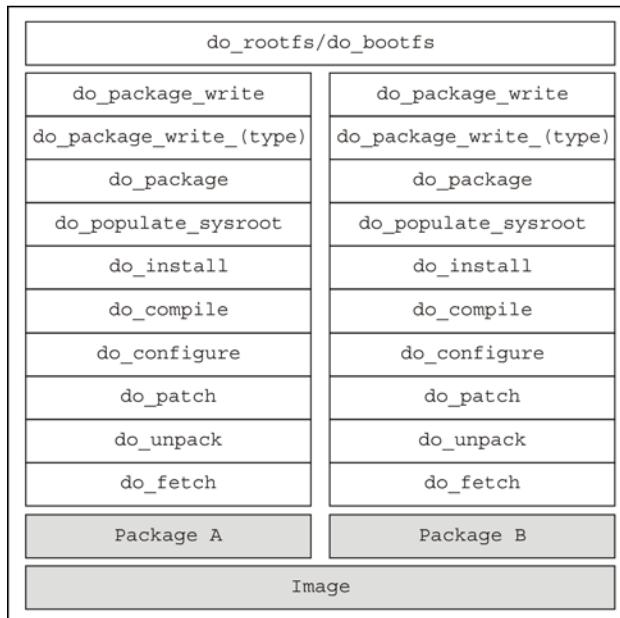
The Yocto Project is sponsored by the Linux Foundation, and offers a starting point for developers of Linux embedded systems who are interested in developing a customized distribution for embedded products in a **hardware-agnostic environment**. The Poky build system represents one of its core components and is also quite complex. At the center of all this lies Bitbake, the engine that powers everything, the tool that processes metadata, downloads corresponding source codes, resolves dependencies, and stores all the necessary libraries and executables inside the build directory accordingly. Poky combines the best from OpenEmbedded with the idea of layering additional software components that could be added or removed from a build environment configuration, depending on the needs of the developer.

Poky is build system that is developed with the idea of keeping simplicity in mind. By default, the configuration for a test build requires very little interaction from the user. Based on the clone made in one of the previous exercises, we can do a new exercise to emphasize this idea:

```
cd poky
source oe-init-build-env ..../build-test
bitbake core-image-minimal
```

As shown in this example, it is easy to obtain a Linux image that can be later used for testing inside a QEMU environment. There are a number of images footprints available that will vary from a shell-accessible minimal image to an LSB compliant image with GNOME Mobile user interface support. Of course, that these base images can be imported in new ones for added functionalities. The layered structure that Poky has is a great advantage because it adds the possibility to extend functionalities and to contain the impact of errors. Layers could be used for all sort of functionalities, from adding support for a new hardware platform to extending the support for tools, and from a new software stack to extended image features. The sky is the limit here because almost any recipe can be combined with another.

All this is possible because of the Bitbake engine, which, after the environment setup and the tests for minimal systems requirements are met, based on the configuration files and input received, identifies the interdependencies between tasks, the execution order of tasks, generates a fully functional cross-compilation environment, and starts building the necessary native and target-specific packages tasks exactly as they were defined by the developer. Here is an example with a list of the available tasks for a package:



More information about Bitbake and its baking process can be found in *Embedded Linux Development with Yocto Project*, by Otavio Salvador and Daiane Angolini.

The metadata modularization is based on two ideas—the first one refers to the possibility of prioritizing the structure of layers, and the second refers to the possibility of not having the need for duplicate work when a recipe needs changes. The layers are overlapping. The most general layer is meta, and all the other layers are usually stacked over it, such as `meta-yocto` with Yocto-specific recipes, machine specific board support packages, and other optional layers, depending on the requirements and needs of developers. The customization of recipes should be done using `bbappend` situated in an upper layer. This method is preferred to ensure that the duplication of recipes does not happen, and it also helps to support newer and older versions of them.

Introduction

An example of the organization of layers is found in the previous example that specified the list of the available tasks for a package. If a user is interested in identifying the layers used by the `test` build setup in the previous exercise that specified the list of the available tasks for a package, the `bblayers.conf` file is a good source of inspiration. If `cat` is done on this file, the following output will be visible:

```
# LAYER_CONF_VERSION is increased each time
build/conf/bblayers.conf
# changes incompatibly
LCONF_VERSION = "6"

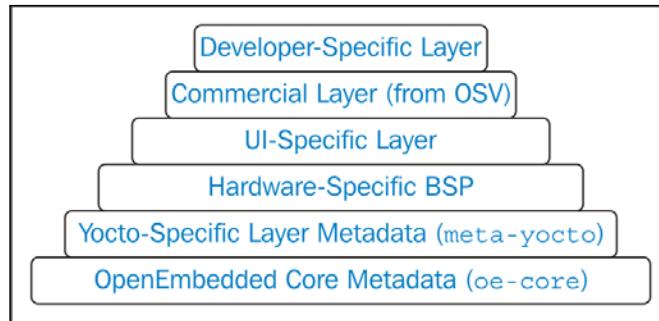
BBPATH = "${TOPDIR}"
BBFILES ?= ""

BBLAYERS ?= " \
/home/alex/workspace/book/poky/meta \
/home/alex/workspace/book/poky/meta-yocto \
/home/alex/workspace/book/poky/meta-yocto-bsp \
"
BBLAYERS_NON_REMOVABLE ?= " \
/home/alex/workspace/book/poky/meta \
/home/alex/workspace/book/poky/meta-yocto \
"
"
```

The complete command for doing this is:

```
cat build-test/conf/bblayers.conf
```

Here is a visual mode for the layered structure of a more generic build directory:



Yocto as a project offers another important feature: the possibility of having an image regenerated in the same way, no matter what factors change on your host machine. This is a very important feature, taking into consideration not only that, in the development process, changes to a number of tools, such as autotools, cross-compiler, *Makefile*, perl, bison, *pkgconfig*, and so on, could occur, but also the fact that parameters could change in the interaction process with regards to a repository. Simply cloning one of the repository branches and applying corresponding patches may not solve all the problems. The solution that the Yocto Project has to these problems is quite simple. By defining parameters prior to any of the steps of the installation as variables and configuration parameters inside recipes, and by making sure that the configuration process is also automated, will minimize the risks of manual interaction are minimized. This process makes sure that image generation is always done as it was the first time.

Since the development tools on the host machine are prone to change, Yocto usually compiles the necessary tools for the development process of packages and images, and only after their build process is finished, the Bitbake build engine starts building the requested packages. This isolation from the developer's machine helps the development process by guaranteeing the fact that updates from the host machine do not influence or affect the processes of generating the embedded Linux distribution.

Another critical point that was elegantly solved by the Yocto Project is represented by the way that the toolchain handles the inclusion of headers and libraries; because this could bring later on not only compilation but also execution errors that are very hard to predict. Yocto resolves these problems by moving all the headers and libraries inside the corresponding *sysroots* directory, and by using the *sysroot* option, the build process makes sure that no contamination is done with the native components. An example will emphasize this information better:

```
ls -l build-test/tmp/sysroots/
total 12K
drwxr-xr-x 8 alex alex 4,0K sep 28 04:17 qemux86/
drwxr-xr-x 5 alex alex 4,0K sep 28 00:48 qemux86-tcbootstrap/
drwxr-xr-x 9 alex alex 4,0K sep 28 04:21 x86_64-linux/
```

```
ls -l build-test/tmp/sysroots/qemux86/
total 24K
drwxr-xr-x 2 alex alex 4,0K sep 28 01:52 etc/
drwxr-xr-x 5 alex alex 4,0K sep 28 04:15 lib/
drwxr-xr-x 6 alex alex 4,0K sep 28 03:51 pkgdata/
drwxr-xr-x 2 alex alex 4,0K sep 28 04:17 sysroot-providers/
drwxr-xr-x 7 alex alex 4,0K sep 28 04:16 usr/
drwxr-xr-x 3 alex alex 4,0K sep 28 01:52 var/
```

The Yocto project contributes to making reliable embedded Linux development and because of its dimensions, it is used for lots of things, ranging from board support packages by hardware companies to new software solutions by software development companies. Yocto is not a perfect tool and it has certain drawbacks:

- Requirements for disk space and machine usage are quite high
- Documentation for advanced usage is lacking
- Tools, such as Autobuilder and Eclipse plug-ins, now have functionality problems

There are also other things that bother developers, such as ptest integration and SDK sysroot's lack of extensibility, but a part of them are solved by the big community behind the project, and until the project shows its limitations, a new one will still need to wait to take its place. Until this happens, Yocto is the framework to use to develop custom embedded Linux distribution or products based in Linux.

Summary

In this chapter, you were presented with the advantages of open source, and examples of how open source helped the Linux kernel, Yocto Project, OpenEmbedded, and Buildroot for the development and growth of projects, such as LTIB and Scratchbox; the lack of open source contribution meant the deprecation and disappearance of them over time. The information presented to you will be in the form of examples, which will give you a clearer idea of the concepts in this book.

In the next chapter, there will be more information on toolchains and its constituent components. Exercises that give you a better idea of toolchains will be generated using both the manual and automatic approach.

2

Cross-compiling

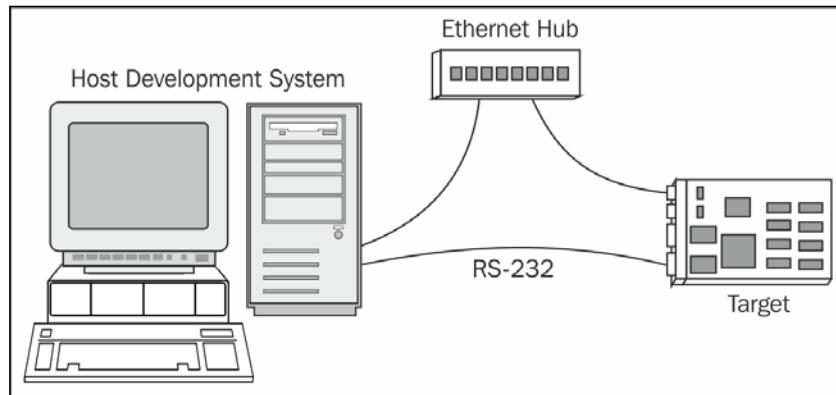
In this chapter, you will learn about toolchains, how to use and customize them, and how code standards apply to them. A toolchain contains a myriad of tools, such as compilers, linkers, assemblers, debuggers, and a variety of miscellaneous utilities that help to manipulate the resulting application binaries. In this chapter, you will learn how to use the GNU toolchain and become familiar with its features. You will be presented with examples that will involve manual configurations, and at the same time, these examples will be moved to the Yocto Project environment. At the end of the chapter, an analysis will be made to identify the similarities and differences between manual and automatic deployment of a toolchain, and the various usage scenarios available for it.

Introducing toolchains

A toolchain represents a compiler and its associated utilities that are used with the purpose of producing kernels, drivers, and applications necessary for a specific target. A toolchain usually contains a set of tools that are usually linked to each other. It consists of `gcc`, `glibc`, `binutils`, or other optional tools, such as a debugger optional compiler, which is used for specific programming languages, such as C++, Ada, Java, Fortran, or Objective-C.

Cross-compiling

Usually a toolchain, which is available on a traditional desktop or server, executes on these machines and produces executables and libraries that are available and can run on the same system. A toolchain that is normally used for an embedded development environment is called a cross toolchain. In this case, programs, such as gcc, run on the host system for a specific target architecture, for which it produces a binary code. This whole process is referred to as cross-compilation, and it is the most common way to build sources for embedded development.



In a toolchain environment, three different machines are available:

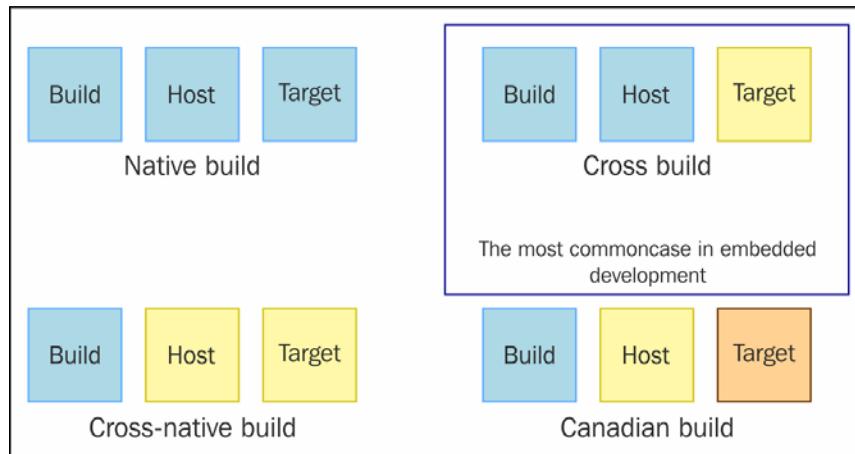
- The build machine that represents the machine on which the toolchain was created
- The host machine that represents the machine on which the toolchain is executed
- The target machine that represents the machine that the toolchain produces a binary code for

These three machines are used to generate four different toolchain build procedures:

- **A native toolchain:** This is usually available on a normal Linux distribution or on your normal desktop system. This is usually compiled and run, and generates code for the same architecture.
- **A cross-native toolchain:** This represents a toolchain built on one system, though it runs and produces a binary code for the target system. A normal use case is when a native gcc is needed on the target system without building it on the target platform.
- **A cross-compilation toolchain:** This is the most widespread toolchain type used for embedded development. It is compiled and run on an architecture type, usually x86, and produces a binary code for the target architecture.

- **A cross-canadian build:** This represents a process that involves building a toolchain on system A. This toolchain is then run on another system, such as B, which produces a binary code for a third system, called C. This is one of the most underused build processes.

The three machines that generate four different toolchain build procedures is described in the following diagram:



Toolchains represent a list of tools that make the existence of most of great projects available today possible. This includes open source projects as well. This diversity would not have been possible without the existence of a corresponding toolchain. This also happens in the embedded world where newly available hardware needs the components and support of a corresponding toolchain for its **Board Support Package (BSP)**.

Toolchain configuration is no easy process. Before starting the search for a prebuilt toolchain, or even building one yourself, the best solution would be to check for a target specific BSP; each development platform usually offers one.

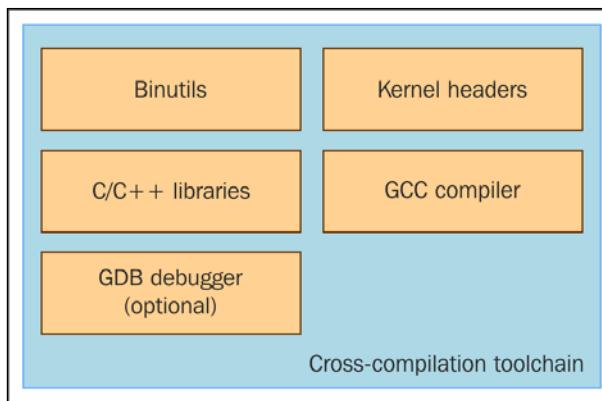
Components of toolchains

The GNU toolchain is a term used for a collection of programming tools under the **GNU Project umbrella**. This suite of tools is what is normally called a **toolchain**, and is used for the development of applications and operating systems. It plays an important role in the development of embedded systems and Linux systems, in particular.

The following projects are included in the GNU toolchain:

- **GNU make:** This represents an automation tool used for compilation and build
- **GNU Compiler Collection (GCC):** This represents a compiler's suite that is used for a number of available programming languages
- **GNU Binutils:** This contains tools, such as linkers, assemblers, and so on - these tools are able to manipulate binaries
- **GNU Bison:** This is a parser generator
- **GNU Debugger (GDB):** This is a code debugging tool
- **GNU m4:** This is an m4 macro processor
- **GNU build system (autotools):** This consists of the following:
 - Autoconf
 - Autoheaders
 - Automake
 - Libtool

The projects included in the toolchain is described in the following diagram:



An embedded development environment needs more than a cross-compilation toolchain. It needs libraries and it should target system-specific packages, such as programs, libraries, and utilities, and host specific debuggers, editors, and utilities. In some cases, usually when talking about a company's environment, a number of servers host target devices, and certain hardware probes are connected to the host through Ethernet or other methods. This emphasizes the fact that an embedded distribution includes a great number of tools, and, usually, a number of these tools require customization. Presenting each of these will take up more than a chapter in a book.

In this book, however, we will cover only the toolchain building components. These include the following:

- `binutils`
- `gcc`
- `glibc` (C libraries)
- kernel headers

I will start by introducing the first item on this list, the **GNU Binutils package**. Developed under the GNU GPL license, it represents a set of tools that are used to create and manage binary files, object code, assembly files, and profile data for a given architecture. Here is a list with the functionalities and names of the available tools for GNU Binutils package:

- The GNU linker, that is `ld`
- The GNU assembler, that is `as`
- A utility that converts addresses into filenames and line numbers, that is `addr2line`
- A utility to create, extract, and modify archives, that is `ar`
- A tool used to listing the symbols available inside object files, that is `nm`
- Copying and translating object files, that is `objcopy`
- Displaying information from object files, that is `objdump`
- Generating an index to for the contents of an archive, that is `ranlib`
- Displaying information from any ELF format object file, that is `readelf`
- Listing the section sizes of an object or archive file, that is `size`
- Listing printable strings from files, that is `strings`
- Discarding the symbols utility that is `strip`
- Filtering or demangle encoded C++ symbols, that is `c++filt`
- Creating files that build use DLLs, that is `dlltool`
- A new, faster, ELF-only linker, which is still in beta testing, that is `gold`
- Displaying the profiling information tool, that is `gprof`
- Converting an object code into an NLM, that is `nlmconv`
- A Windows-compatible message compiler, that is `windmc`
- A compiler for Windows resource files, that is `windres`

The majority of these tools use the **Binary File Descriptor (BFD)** library for low-level data manipulation, and also, many of them use the `opcode` library to assemble and disassemble operations.



Useful information about binutils can be found at <http://www.gnu.org/software/binutils/>.



In the toolchain generation process, the next item on the list is represented by kernel headers, and are needed by the C library for interaction with the kernel. Before compiling the corresponding C library, the kernel headers need to be supplied so that they can offer access to the available system calls, data structures, and constants definitions. Of course, any C library defines sets of specifications that are specific to each hardware architecture; here, I am referring to **application binary interface (ABI)**.

An application binary interface (ABI) represents the interface between two modules. It gives information on how functions are called and the kind of information that should be passed between components or to the operating system. Referring to a book, such as *The Linux Kernel Primer*, will do you good, and in my opinion, is a complete guide for what the ABI offers. I will try to reproduce this definition for you.

An ABI can be seen as a set of rules similar to a protocol or an agreement that offers the possibility for a linker to put together compiled modules into one component without the need of recompilation. At the same time, an ABI describes the binary interface between these components. Having this sort of convention and conforming to an ABI offers the benefits of linking object files that could have been compiled with different compilers.

It can be easily seen from both of these definitions that an ABI is dependent on the type of platform, which can include physical hardware, some kind of virtual machine, and so on. It may also be dependent on the programming language that is used and the compiler, but most of it depends on the platform.

The ABI presents how the generated codes operate. The code generation process must also be aware of the ABI, but when coding in a high-level language, attention given to the ABI is rarely a problem. This information could be considered as necessary knowledge to specify some ABI related options.

As a general rule, ABI must be respected for its interaction with external components. However, with regard to interaction with its internal modules, the user is free to do whatever he or she wants. Basically, they are able to reinvent the ABI and form their own dependence on the limitations of the machine. The simple example here is related to various citizens who belong to their own country or region, because they learned and know the language of that region since they were born. Hence, they are able to understand one another and communicate without problems. For an external citizen to be able to communicate, he or she will need to know the language of a region, and being in this community seems natural, so it will not constitute a problem. Compilers are also able to design their own custom calling conventions where they know the limitations of functions that are called within a module. This exercise is typically done for optimization reasons. However, this can be considered an abuse of the ABI term.

The kernel in reference to a user space ABI is backward compatible, and it makes sure that binaries are generated using older kernel header versions, rather than the ones available on the running kernel, will work best. The disadvantages of this are represented by the fact that new system calls, data structures, and binaries generated with a toolchain that use newer kernel headers, might not work for newer features. The need for the latest kernel headers can be justified by the need to have access to the latest kernel features.

The GNU Compiler Collection, also known as GCC, represents a compiler system that constitutes the key component of the GNU toolchain. Although it was originally named the GNU C Compiler, due to the fact that it only handled the C programming language, it soon began to represent a collection of languages, such as C, C++, Objective C, Fortran, Java, Ada, and Go, as well as the libraries for other languages (such as `libstdc++`, `libgcj`, and so on).

It was originally written as the compiler for the GNU operating system and developed as a 100 percent free software. It is distributed under the GNU GPL. This helped it extend to its functionalities across a wide variety of architectures, and it played an important role in the growth of open source software.

The development of GCC started with the effort put in by Richard Stallman to bootstrap the GNU operating system. This quest led Stallman to write his own compiler from scratch. It was released in 1987, with Stallman as the author and other as contributors to it. By 1991, it had already reached a stable phase, but it was unable to include improvements due to its architectural limitations. This meant that the starting point for work on GCC version 2 had begun, but it did not take long until the need for development of new language interfaces started to appear in it as well, and developers started doing their own forks of the compiler source code. This fork initiative proved to be very inefficient, and because of the difficulty of accepting the code procedure, working on it became really frustrating.

This changed in 1997, when a group of developers gathered as the **Experimental/Enhanced GNU Compiler System (EGCS)** workgroup started merging several forks in one project. They had so much success in this venture, and gathered so many features, that they made **Free Software Foundation (FSF)** halt their development of GCC version 2 and appointed EGCS the official GCC version and maintainers by April 1999. They united with each other with the release of GCC 2.95. More information on the history and release history of the GNU Compiler Collection can be found at <https://www.gnu.org/software/gcc/releases.html> and http://en.wikipedia.org/wiki/GNU_Compiler_Collection#Revision_history.

The GCC interface is similar to the Unix convention, where users call a language-specific driver, which interprets arguments and calls a compiler. It then runs an assembler on the resulting outputs and, if necessary, runs a linker to obtain the final executable. For each language compiler, there is a separate program that performs the source code read.

The process of obtaining an executable from source code has some execution steps. After the first step, an abstract syntax tree is generated and, in this stage, compiler optimization and static code analysis can be applied. The optimizations and static code analysis can be both applied on architecture-independent **GIMPLE** or its superset **GENERIC** representation, and also on architecture-dependent **Register Transfer Language (RTL)** representation, which is similar to the LISP language. The machine code is generated using pattern-matching algorithm, which was written by Jack Davidson and Christopher Fraser.

GCC was initially written almost entirely in C language, although the Ada frontend is written mostly in Ada language. However, in 2012, the GCC committee announced the use of C++ as an implementation language. The GCC library could not be considered finished as an implementation language, even though its main activities include adding new languages support, optimizations, improved runtime libraries, and increased speed for debugging applications.

Each available frontend generated a tree from the given source code. Using this abstract tree form, different languages can share the same backend. Initially, GCC used **Look-Ahead LR (LALR)** parsers, which were generated using Bison, but over time, it moved on to recursive-descendent parsers for C, C++, and Objective-C in 2006. Today, all available frontends use handwritten recursive-descendent parsers.

Until recently, the syntax tree abstraction of a program was not independent of a target processor, because the meaning of the tree was different from one language frontend to the other, and each provided its own tree syntax. All this changed with the introduction of GENERIC and GIMPLE architecture-independent representations, which were introduced with the GCC 4.0 version.

GENERIC is a more complex intermediate representation, while GIMPLE is a simplified GENERIC and targets all the frontends of GCC. Languages, such as C, C++ or Java frontends, directly produce GENERIC tree representations in the frontend. Others use different intermediate representations that are then parsed and converted to GENERIC representations.

The GIMPLE transformation represents complex expressions that are split into a three address code using temporary variables. The GIMPLE representation was inspired by the SIMPLE representation used on the McCAT compiler for simplifying the analysis and optimization of programs.

The middle stage representation of GCC involves code analysis and optimization, and works independently in terms of a compiled language and the target architecture. It starts from the GENERIC representation and continues to the **Register Transfer Language (RTL)** representation. The optimization mostly involves jump threading, instruction scheduling, loop optimization, sub expression elimination, and so on. The RTL optimizations are less important than the ones done through GIMPLE representations. However, they include dead code elimination, global value numbering, partial redundancy elimination, sparse conditional constant propagation, scalar replacement of aggregates, and even automatic vectorization or automatic parallelization.

The GCC backend is mainly represented by preprocessor macros and specific target architecture functions, such as endianness definitions, calling conventions, or word sizes. The initial stage of the backend uses these representations to generate the RTL; this suggests that although GCC's RTL representation is nominally processor-independent, the initial processing of abstract instructions is adapted for each specific target.

A machine-specific description file contains RTL patterns, also code snippets, or operand constraints that form a final assembly. In the process of RTL generation, the constraints of the target architecture are verified. To generate an RTL snippet, it must match one or a number RTL patterns from the machine description file, and at the same time also satisfy the limitations for these patterns. If this is not done, the process of conversion for the final RTL into machine code would be impossible. Toward the end of compilation, the RTL representation becomes a strict form. Its representation contains a real machine register correspondence and a template from the target's machine description file for each instruction reference.

As a result, the machine code is obtained by calling small snippets of code, which are associated with corresponding patterns. In this way, instructions are generated from target instruction sets. This process involves the usage of registers, offsets, and addresses from the reload phase.



More information about a GCC compiler can be found at <http://gcc.gnu.org/> or http://en.wikipedia.org/wiki/GNU_Compiler_Collection.



The last element that needs to be introduced here is the C library. It represents the interface between a Linux kernel and applications used on a Linux system. At the same time, it offers aid for the easier development of applications. There are a couple of C libraries available in this community:

- glibc
- eglibc
- Newlib
- bionic
- musl
- uClibc
- dietlibc
- Klibc

The choice of the C library used by the GCC compiler will be executed in the toolchain generation phase, and it will be influenced not only by the size and application support offered by the libraries, but also by compliance of standards, completeness, and personal preference.

Delving into C libraries

The first library that we'll discuss here is the `glibc` library, which is designed for performance, compliance of standards, and portability. It was developed by the Free Software Foundation for the GNU/Linux operating system and is still present today on all GNU/Linux host systems that are actively maintained. It was released under the GNU Lesser General Public License.

The `glibc` library was initially written by Roland McGrath in the 1980s and it continued to grow until the 1990s when the Linux kernel forked `glibc`, calling it `Linux libc`. It was maintained separately until January 1997 when the Free Software Foundation released `glibc 2.0`. The `glibc 2.0` contained so many features that it did not make any sense to continue the development of `Linux libc`, so they discontinued their fork and returned to using `glibc`. There are changes that are made in `Linux libc` that were not merged into `glibc` because of problems with the authorship of the code.

The `glibc` library is quite large in terms of its dimensions and isn't a suitable fit for small embedded systems, but it provides the functionality required by the **Single UNIX Specification (SUS)**, POSIX, ISO C11, ISO C99, Berkeley Unix interfaces, System V Interface Definition, and the X/Open Portability Guide, Issue 4.2, with all its extensions common with X/Open System Interface compliant systems along with X/Open UNIX extensions. In addition to this, GLIBC also provides extensions that have been deemed useful or necessary while developing GNU.

The next C library that I'm going to discuss here is the one that resides as the main C library used by the Yocto Project until version 1.7. Here, I'm referring to the `eglibc` library. This is a version of `glibc` optimized for the usage of embedded devices and is, at the same time, able to preserve the compatibility standards.

Since 2009, Debian and a number of its derivations chose to move from the GNU C Library to `eglibc`. This might be because there is a difference in licensing between GNU LGPL and `eglibc`, and this permits them to accept patches that `glibc` developers may reject. Since 2014, the official `eglibc` homepage states that the development of `eglibc` was discontinued because `glibc` had also moved to the same licensing, and also, the release of Debian Jessie meant that it had moved back to `glibc`. This also happened in the case of Yocto support when they also decided to make `glibc` their primary library support option.

The `newlib` library is another C library developed with the intention of being used in embedded systems. It is a conglomerate of library components under free software licenses. Developed by Cygnus Support and maintained by Red Hat, it is one of the preferred versions of the C library used for non-Linux embedded systems.

The newlib system calls describe the usage of the C library across multiple operation systems, and also on embedded systems that do not require an operating system. It is included in commercial GCC distributions, such as Red Hat, CodeSourcery, Attolic, KPIT and others. It also supported by architecture vendors that include ARM, Renesas, or Unix-like environments, such as Cygwin, and even proprietary operating systems of the Amiga personal computer.

By 2007, it also got support from the toolchain maintainers of Nintendo DS, PlayStation, portable SDK Game Boy Advance systems, Wii, and GameCube development platforms. Another addition was made to this list in 2013 when Google Native Client SDK included newlib as their primary C library.

Bionic is a derivate of the BSD C library developed by Google for Android based on the Linux kernel. Its development is independent of Android code development. It is licensed as 3-clause BSD license and its goals are publically available. These include the following:

- **Small size:** Bionic is smaller in size compared to glibc
- **Speed:** This has designed CPUs that work at low frequencies
- **BSD license:** Google wished to isolate Android apps from GPL and LGPL licenses, and this is the reason it moved to a non-copyleft license which are as follows:
 - Android is based on a Linux kernel which is based on a GPLv2 license
 - glibc is based on LGPL, which permits the linking of dynamic proprietary libraries but not with static linking

It also has a list of restrictions compared to glibc, as follows:

- It does not include C++ exception handling, mainly because most the code used for Android is written in Java.
- It does not have wide character support.
- It does not include a Standard Template library, although it can be included manually.
- It functions within Bionic POSIX and even system call headers are wrappers or stubs for Android -specific functions. This may lead to odd behavior sometimes.
- When Android 4.2 released, it included support for glibc FORTIFY_SOURCE features. These features are very often used in Yocto, and embedded systems in general, but are only present in the gcc version for Android devices with ARM processors.

The next C library that will be discussed is `musl`. It is a C library intended for use with Linux operating systems for embedded and mobile systems. It has a MIT license and was developed with the idea of having a clean, standard-compliant `libc`, which is time efficient, since it's been developed from scratch. As a C library, it is optimized for the linking of static libraries. It is compatible with C99 standard and POSIX 2008, and implements Linux, `glibc`, and BSD non-standard functions.

Next, we'll discuss `uClibc`, which is a C standard library designed for Linux embedded systems and mobile devices. Although initially developed for `μClinux` and designed for microcontrollers, it gathered track and became the weapon of choice for anyone who's has limited space on their device. This has become popular due to the following reasons:

- It focuses on size rather than performance
- It has a GNU Lesser General Public License (LGPL) free license
- It is much smaller the `glibc` and reduces compilation time
- It has high configurability due to the fact that many of its features can be enabled using a `menuconfig` interface similar to the one available on packages, such as Linux kernel, U-Boot, or even BusyBox

The `uClibc` library also has another quality that makes it quite useful. It introduces a new ideology and, because of this, the C library does not try to support as many standards as possible. However, it focuses on embedded Linux and consists of the features necessary for developers who face the limitation of available space. Due to this reason, this library was written from scratch, and even though it has its fair share of limitations, `uClibc` is an important alternative to `glibc`. If we take into consideration the fact that most of the features used from C libraries are present in it, the final size is four times smaller, and WindRiver, MontaVista, and TimeSys are active maintainers of it.

The `dietlibc` library is a standard C library that was developed by Felix von Leitner and released under the GNU GPL v2 license. Although it also contains some commercial licensed components, its design was based on the same idea as `uClibc`: the possibility of compiling and linking software while having the smallest size possible. It has another resemblance to `uClibc`; it was developed from scratch and has only implemented the most used and known standard functions. Its primary usage is mainly in the embedded devices market.

The last in the C libraries list is the `klibc` standard C library. It was developed by H. Peter Anvin and it was developed to be used as part of the early user space during the Linux startup process. It is used by the components that run the the kernel startup process but aren't used in the kernel mode and, hence, they do not have access to the standard C library.

The development of `klibc` started in 2002 as an initiative to remove the Linux initialization code outside a kernel. Its design makes it suitable for usage in embedded devices. It also has another advantage: it is optimized for small size and correctness of data. The `klibc` library is loaded during the Linux startup process from `initramfs` (a temporary Ram filesystem) and is incorporated by default into `initramfs` using the `mkintramfs` script for Debian and Ubuntu-based filesystems. It also has access to a small set of utilities, such as `mount`, `mkdir`, `dash`, `mknod`, `fstype`, `nfsmount`, `run-init` and so on, which are very useful in the early init stage.



More information on `initramfs` can be found using the kernel documentation at <https://www.kernel.org/doc/Documentation/filesystems/ramfs-rootfs-initramfs.txt>.



The `klibc` library is licensed under GNU GPL since it uses some components from the Linux kernel, so, as a whole, it is visible as a GPL licensed software, limiting its applicability in commercial embedded software. However, most of the source code of libraries is written under the BSD license.

Working with toolchains

When generating a toolchain, the first thing that needs to be done is the establishment of an ABI used to generate binaries. This means that the kernel needs to understand this ABI and, at the same time, all the binaries in the system need to be compiled with the same ABI.

When working with the GNU toolchain, a good source of gathering information and understanding the ways in which work is done with these tools is to consult the GNU coding standards. The coding standard's purposes are very simple: to make sure that the work with the GNU ecosystem is performed in a clean, easy, and consistent manner. This is a guideline that needs to be used by people interested in working with GNU tools to write reliable, solid, and portable software. The main focus of the GNU toolchain is represented by the C language, but the rules applied here are also very useful for any programming languages. The purpose of each rule is explained by making sure that the logic behind the given information is passed to the reader.

The main language that we will be focusing on will also be the C programming language. With regard to the GNU coding standard compatibility regarding libraries for GNU, exceptions or utilities and their compatibility should be very good when compared with standards, such as the ones from Berkeley Unix, Standard C, or POSIX. In case of conflicts in compatibility, it is very useful to have compatibility modes for that programming language.

Standards, such as POSIX and C, have a number of limitations regarding the support for extensions - however, these extensions could still be used by including a `-posix`, `-ansi`, or `-compatible` option to disable them. In case the extension offers a high probability of breaking a program or script by being incompatible, a redesign of its interface should be made to ensure compatibility.

A large number of GNU programs suppress the extensions that are known to cause conflict with POSIX if the `POSIXLY_CORRECT` environment variable is defined. The usage of user defined features offers the possibility for interchanging GNU features with other ones totally different, better, or even use a compatible feature. Additional useful features are always welcomed.

If we take a quick look at the GNU Standard documentation, some useful information can be learned from it:

It is better to use the `int` type, although you might consider defining a narrower data type. There are, of course, a number of special cases where this could be hard to use. One such example is the `dev_t` system type, because it is shorter than `int` on some machines and wider on others. The only way to offer support for non-standard C types involves checking the width of `dev_t` using `Autoconf` and, after this, choosing the argument type accordingly. However, it may not worth the trouble.

For the GNU Project, the implementation of an organization's standard specifications is optional, and this can be done only if it helps the system by making it better overall. In most situations, following published standards fits well within a users needs because their programs or scripts could be considered more portable. One such example is represented by the GCC, which implements almost all the features of Standard C, as the standard requires. This offers a great advantage for the developers of the C program. This also applies to GNU utilities that follow POSIX.2 specifications.

There are also specific points in the specifications that are not followed, but this happens with the sole reason of making the GNU system better for users. One such example would be the fact that the Standard C program does not permit extensions to C, but, GCC implements many of them, some being later embraced by the standard. For developers interested in outputting an error message as *required* by the standard, the `--pedantic` argument can be used. It is implemented with a view to making sure that GCC fully implements the standard.

The POSIX.2 standard mentions that commands, such as `du` and `df`, should output sizes in units of 512 bytes. However, users want units of 1KB and this default behavior is implemented. If someone is interested in having the behavior requested by POSIX standard, they would need to set the `POSIXLY_CORRECT` environment variable.

Another such example is represented by the GNU utilities, which don't always respect the POSIX.2 standard specifications when referring to support for long named command-line options or intermingling of options with arguments. This incompatibility with the POSIX standard is very useful in practice for developers. The main idea here is not to reject any new feature or remove an older one, although a certain standard mentions it as deprecated or forbidden.



For more information regarding the GNU Coding Standards, refer to
https://www.gnu.org/prep/standards/html_node/.



Advice on robust programming

To make sure that you write robust code, a number of guidelines should be mentioned. The first one refers to the fact that limitations should not be used for any data structure, including files, file names, lines, and symbols, and especially arbitrary limitations. All data structures should be dynamically allocated. One of the reasons for this is represented by the fact that most Unix utilities silently truncate long lines; GNU utilities do not do these kind of things.

Utilities that are used to read files should avoid dropping null characters or nonprinting characters. The exception here is when these utilities, that are intended for interfacing with certain types of printers or terminals, are unable to handle the previously mentioned characters. The advice that I'd give in this case would be to try and make programs work with a UTF-8 character set, or other sequences of bytes used to represent multibyte characters.

Make sure that you check system calls for error return values; the exception here is when a developer wishes to ignore the errors. It would be a good idea to include the system error text from `strerror`, `perror`, or equivalent error handling functions, in error messages that result from a crashed on system call, adding the name of the source code file, and also the name of the utility. This is done to make sure that the error message is easy to read and understand by anyone involved in the interaction with the source code or the program.

Check the return value for `malloc` or `realloc` to verify if they've returned zero. In case `realloc` is used in order to make a block smaller in systems that approximate block dimensions to powers of 2, `realloc` may have a different behavior and get a different block. In Unix, when `realloc` has a bug, it destroys the storage block for a zero return value. For GNU, this bug does not occur, and when it fails, the original block remains unchanged. If you want to run the same program on Unix and do not want to lose data, you could check if the bug was resolved on the Unix system or use the `malloc` GNU.

The content of the block that was freed is not accessible to alter or for any other interactions from the user. This can be done before calling free.

When a `malloc` command fails in a noninteractive program, we face a fatal error. In case the same situation is repeated, but, this time, an interactive program is involved, it would be better to abort the command and return to the read loop. This offers the possibility to free up virtual memory, kill other processes, and retry the command.

To decode arguments, the `getopt_long` option can be used.

When writing static storage during program execution, use C code for its initialization. However, for data that will not be changed, reserve C initialized declarations.

Try to keep away from low-level interfaces to unknown Unix data structures - this could happen when the data structure do not work in a compatible fashion. For example, to find all the files inside a directory, a developer could use the `readdir` function, or any high-level interface available function, since these do not have compatibility problems.

For signal handling, use the BSD variant of `signal` and the POSIX `sigaction` function. The USG `signal` interface is not the best alternative in this case. Using POSIX signal functions is nowadays considered the easiest way to develop a portable program. However, the use of one function over another is completely up to the developer.

For error checks that identify impossible situations, just abort the program, since there is no need to print any messages. These type of checks bear witness to the existence of bugs. To fix these bugs, a developer will have to inspect the available source code and even start a debugger. The best approach to solve this problem would be to describe the bugs and problems using comments inside the source code. The relevant information could be found inside variables after examining them accordingly with a debugger.

Do not use a count of the encountered errors in a program as an exit status. This practice is not the best, mostly because the values for an exit status are limited to 8 bits only, and an execution of the executable might have more than 255 errors. For example, if you try to return exit status 256 for a process, the parent process will see a status of zero and consider that the program finished successfully.

If temporary files are created, checking that the `TMPDIR` environment variable would be a good idea. If the variable is defined, it would be wise to use the `/tmp` directory instead. The use of temporary files should be done with caution because there is the possibility of security breaches occurring when creating them in world-writable directories. For C language, this can be avoided by creating temporary files in the following manner:

```
fd = open (filename, O_WRONLY | O_CREAT | O_EXCL, 0600);
```

This can also be done using the `mkstemp` function, which is made available by GnuLib.

For a bash environment, use the `noclobber` environment variable, or the `set -C` short version, to avoid the previously mentioned problem. Furthermore, the `mkttemp` available utility is altogether a better solution for making a temporary file a shell environment; this utility is available in the GNU Coreutils package.



More information about GNU C Standards can be found at <https://www.gnu.org/prep/standards/standards.html>.



Generating the toolchain

After the introduction of the packages that comprise a toolchain, this section will introduce the steps needed to obtain a custom toolchain. The toolchain that will be generated will contain the same sources as the ones available inside the Poky dizzy branch. Here, I am referring to the `gcc` version 4.9, `binutils` version 2.24, and `glibc` version 2.20. For Ubuntu systems, there are also shortcuts available. A generic toolchain can be installed using the available package manager, and there are also alternatives, such as downloading custom toolchains available inside Board Support Packages, or even from third parties, including CodeSourcery and Linaro. More information on toolchains can be found at <http://elinux.org/Toolchains>. The architecture that will be used as a demo is an ARM architecture.

The toolchain build process has eight steps. I will only outline the activities required for each one of them, but I must mention that they are all automatized inside the Yocto Project recipes. Inside the Yocto Project section, the toolchain is generated without notice. For interaction with the generated toolchain, the simplest task would be to call `meta-ide-support`, but this will be presented in the appropriate section as follows:

- **The setup:** This represents the step in which top-level build directories and source subdirectories are created. In this step, variables such as `TARGET`, `SYSROOT`, `ARCH`, `COMPILER`, `PATH`, and others are defined.
- **Geting the sources:** This represents the step in which packages, such as `binutils`, `gcc`, `glibc`, `linux kernel headers`, and various patches are made available for use in later steps.

- **GNU Binutils setup** - This represents the steps in which the interaction with the `binutils` package is done, as shown here:
 - Unzip the sources available from the corresponding release
 - Patch the sources accordingly, if this applies
 - Configure, the package accordingly
 - Compile the sources
 - Install the sources in the corresponding location
- **Linux kernel headers setup:** This represents the steps in which the interaction with the Linux kernel sources is presented, as shown here:
 - Unzip the kernel sources.
 - Patch the kernel sources, if this applies.
 - Configure the kernel for the selected architecture. In this step, the corresponding kernel config file is generated. More information about Linux kernel will be presented in *Chapter 4, Linux Kernel*.
 - Compile the Linux kernel headers and copy them in the corresponding location.
 - Install the headers in the corresponding locations.
- **Glibc headers setup:** This represents the steps used to setting the `glibc` build area and installation headers, as shown here:
 - Unzip the `glibc` archive and headers files
 - Patch the sources, if this applies
 - Configure the sources accordingly enabling the `-with-headers` variable to link the libraries to the corresponding Linux kernel headers
 - Compile the `glibc` headers files
 - Install the headers accordingly
- **GCC first stage setup:** This represents the step in which the C runtime files, such as `crti.o` and `crtn.o`, are generated:
 - Unzip the `gcc` archive
 - Patch the `gcc` sources if necessary
 - Configure the sources enabling the needed features
 - Compile the C runtime components
 - Install the sources accordingly

- **Build the glibc sources:** This represents the step in which the glibc sources are built and the necessary ABI setup is done, as shown here:
 - Configure the glibc library by setting the `mabi` and `march` variables accordingly
 - Compile the sources
 - Install the glibc accordingly
- **GCC second stage setup:** This represents the final setup phase in which the toolchain configuration is finished, as shown here:
 - Configure the gcc sources
 - Compile the sources
 - Install the binaries in the corresponding location

After these steps are performed, a toolchain will be available for the developer to use. The same strategy and build procedure steps is followed inside the Yocto Project.

The Yocto Project reference

As I have mentioned, the major advantage and available feature of the Yocto Project environment is represented by the fact that a Yocto Project build does not use the host available packages, but builds and uses its own packages. This is done to make sure that a change in the host environment does not influence its available packages and that builds are made to generate a custom Linux system. A toolchain is one of the components because almost all packages that are constituents of a Linux distribution need the usage of toolchain components.

The first step for the Yocto Project is to identify the exact sources and packages that will be combined to generate the toolchain that will be used by later built packages, such as U-Boot bootloader, kernel, BusyBox and others. In this book, the sources that will be discussed are the ones available inside the dizzy branch, the latest poky 12.0 version, and the Yocto Project version 1.7. The sources can be gathered using the following command:

```
git clone -b dizzy http://git.yoctoproject.org/git/poky
```

Gathering the sources and investigating the source code, we identified a part of the packages mentioned and presented in the preceding headings, as shown here:

```
cd poky
find ./ -name "gcc"
./meta/recipes-devtools/gcc
find ./ -name "binutils"
./meta/recipes-devtools/binutils
./meta/recipes-devtools/binutils/binutils
find ./ -name "glibc"
./meta/recipes-core/glibc
./meta/recipes-core/glibc/glibc
$ find ./ -name "uclibc"
./meta-yocto-bsp/recipes-core/uclibc
./meta-yocto-bsp/recipes-core/uclibc/uclibc
./meta/recipes-core/uclibc
```

The GNU CC and GCC C compiler package, which consists of all the preceding packages, is split into multiple fractions, each one with its purpose. This is mainly because each one has its purpose and is used with different scopes, such as `sdk` components. However, as I mentioned in the introduction of this chapter, there are multiple toolchain build procedures that need to be assured and automated with the same source code. The available support inside Yocto is for gcc 4.8 and 4.9 versions. A quick look at the `gcc` available recipes shows the available information:

```
meta/recipes-devtools/gcc/
└── gcc-4.8
└── gcc_4.8.bb
└── gcc-4.8.inc
└── gcc-4.9
└── gcc_4.9.bb
└── gcc-4.9.inc
└── gcc-common.inc
└── gcc-configure-common.inc
```

Cross-compiling

```
|── gcc-cross_4.8.bb
|── gcc-cross_4.9.bb
|── gcc-cross-canadian_4.8.bb
|── gcc-cross-canadian_4.9.bb
|── gcc-cross-canadian.inc
|── gcc-cross.inc
|── gcc-cross-initial_4.8.bb
|── gcc-cross-initial_4.9.bb
|── gcc-cross-initial.inc
|── gcc-crosssdk_4.8.bb
|── gcc-crosssdk_4.9.bb
|── gcc-crosssdk.inc
|── gcc-crosssdk-initial_4.8.bb
|── gcc-crosssdk-initial_4.9.bb
|── gcc-crosssdk-initial.inc
|── gcc-multilib-config.inc
|── gcc-runtime_4.8.bb
|── gcc-runtime_4.9.bb
|── gcc-runtime.inc
|── gcc-target.inc
|── libgcc_4.8.bb
|── libgcc_4.9.bb
|── libgcc-common.inc
|── libgcc.inc
|── libgcc-initial_4.8.bb
|── libgcc-initial_4.9.bb
|── libgcc-initial.inc
|── libgfortran_4.8.bb
|── libgfortran_4.9.bb
└── libgfortran.inc
```

The GNU Binutils package represents the binary tools collection, such as GNU Linker, GNU Assembler, addr2line, ar, nm, objcopy, objdump, and other tools and related libraries. The Yocto Project offers support for the Binutils version 2.24, and is also dependent on the available toolchain build procedures, as it can be viewed from the inspection of the source code:

```
meta/recipes-devtools/binutils/
|__ binutils
|__ binutils_2.24.bb
|__ binutils-2.24.inc
|__ binutils-cross_2.24.bb
|__ binutils-cross-canadian_2.24.bb
|__ binutils-cross-canadian.inc
|__ binutils-cross.inc
|__ binutils-crosssdk_2.24.bb
└__ binutils.inc
```

The last components is represented by C libraries that are present as components inside the Poky dizzy branch. There are two C libraries available that can be used by developers. The first one is represented by the GNU C library, also known as `glibc`, which is the most used C library in Linux systems. The sources for `glibc` package can be viewed here:

```
meta/recipes-core/glibc/
|__ cross-localedef-native
|__ cross-localedef-native_2.20.bb
|__ glibc
|__ glibc_2.20.bb
|__ glibc-collateral.inc
|__ glibc-common.inc
|__ glibc.inc
|__ glibc-initial_2.20.bb
|__ glibc-initial.inc
|__ glibc-ld.inc
|__ glibc-locale_2.20.bb
|__ glibc-locale.inc
|__ glibc-mtrace_2.20.bb
|__ glibc-mtrace.inc
|__ glibc-options.inc
|__ glibc-package.inc
|__ glibc-scripts_2.20.bb
|__ glibc-scripts.inc
```

```
|── glibc-testing.inc
|── ldconfig-native-2.12.1
|── ldconfig-native_2.12.1.bb
└── site_config
```

From these sources, the same location also includes tools, such as `ldconfig`, a standalone native dynamic linker for runtime dependencies and a binding and cross locale generation tool. In the other C library, called `uClibc`, as previously mentioned, a library designed for embedded systems has fewer recipes, as it can be viewed from the Poky source code:

```
meta/recipes-core/uclibc/
├── site_config
├── uclibc-config.inc
├── uclibc-git
├── uclibc_git.bb
├── uclibc-git.inc
├── uclibc.inc
├── uclibc-initial_git.bb
└── uclibc-package.inc
```

The `uClibc` is used as an alternative to `glibc` C library because it generates smaller executable footprints. At the same time, `uClibc` is the only package from the ones presented in the preceding list that has a `bbappend` applied to it, since it extends the support for two machines, `genericx86-64` and `genericx86`. The change between `glibc` and `uClibc` can be done by changing the `TCLIBC` variable to the corresponding variable in this way: `TCLIBC = "uClibc"`.

As mentioned previously, the toolchain generation process for the Yocto Project is simpler. It is the first task that is executed before any recipe is built using the Yocto Project. To generate the cross-toolchain inside using Bitbake, first, the `bitbake meta-ide-support` task is executed. The task can be executed for the `qemuarm` architecture, for example, but it can, of course, be generated in a similar method for any given hardware architecture. After the task finishes the execution process, the toolchain is generated and it populates the build directory. It can be used after this by sourcing the `environment-setup` script available in the `tmp` directory:

```
cd poky
source oe-init-build-env ../build-test
```

Set the `MACHINE` variable to the value `qemuarm` accordingly inside the `conf/local.conf` file:

```
bitbake meta-ide-support  
source tmp/environment-setup
```

The default C library used for the generation of the toolchain is `glibc`, but it can be changed according to the developer's need. As seen from the presentation in the previous section, the toolchain generation process inside the Yocto Project is very simple and straightforward. It also avoids all the trouble and problems involved in the manual toolchain generation process, making it very easy to reconfigure also.

Summary

In this chapter, you were presented with the necessary information needed to understand the constituent components of a Linux toolchain, and the steps undertaken by developers to work or configure a Linux toolchain that is specific for a board or architecture. You were also presented information on the packages available inside the Yocto Project sources, and how the processes defined inside the Yocto Project are very similar to the ones already used outside of the Yocto Project context.

In the next chapter, we will breeze through information on bootloaders, with special emphasis given to U-Boot bootloader. You will also be given information on a boot sequence and a board's configurations inside the U-Boot sources.

3

Bootloaders

In this chapter, you will be presented with one of the most important components necessary for using a Linux system in an embedded environment. Here, I am referring to the bootloader, a piece of software that offers the possibility of initializing a platform and making it ready to boot a Linux operating system. In this chapter, the benefits and roles of bootloaders will be presented. This chapter mainly focuses on the U-Boot bootloaders, but readers are encouraged to have a look at others, such as Barebox, RedBoot, and so on. All these bootloaders have their respective features and there isn't one in particular that suits every need; therefore, experimentation and curiosity are welcome when this chapter. You have already been introduced to the the Yocto Project reference in the last chapter; hence, you will now be able to understand how this development environment works with various bootloaders, and especially the ones available inside a **Board Support Package (BSP)**.

The main purpose of this chapter is to present the main properties of embedded bootloaders and firmware, their booting mechanisms, and the problems that appear when firmware is updated or modified. We will also discuss the problems related to safety, installation, or fault tolerance. With regard to bootloader and firmware notions, we have multiple definitions available and a number of them refer to traditional desktop systems, which we are not interested in.

A firmware usually represents a fixed and small program that is used on a system to control hardware. It performs low-level operations and is usually stored on flash, ROM, EPROM, and so on. It is not changed very often. Since there have been situations where this term has confused people and was sometimes used only to define hardware devices or represent data and its instructions, it was avoided altogether. It represents a combination of the two: computer data and information, along with the hardware device combined in a read-only piece of software available on the device.

The bootloader represents the piece of software that is first executed during system initialization. It is used to load, decompress, and execute one or more binary applications, such as a Linux kernel or a root filesystem. Its role involves adding the system in a state where it can execute its primary functions. This is done after loading and starting the correct binary applications that it receives or has already saved on the internal memory. Upon initializing, the hardware bootloader may need to initialize the **phase-locked loop (PLL)**, set the clocks, or enable access to the RAM memory and other peripherals. However, these initializations are done on a basic level; the rest are done by kernels drivers and other applications.

Today, a number of bootloaders are available. Due to limited space available for this topic, and also the fact that their number is high, we will only discuss the most popular ones. U-Boot is one of the most popular bootloaders available for architectures, such as PowerPC, ARM, MIPS, and others. It will constitute the primary focus of this chapter.

The role of the bootloader

The first time that electricity runs into a development board processor, a great number of hardware components need to be prepared before running a program. For each architecture, hardware manufacturer, and even processor, this initialization process is different. In most cases, it involves a set of configurations and actions are different for a variety of processors and ends up fetching the bootstrap code from a storage device available in the proximity of the processor. This storage device is usually a flash memory and the bootstrap code is the first stage of the bootloader, and the one that initializes the processor and relevant hardware peripherals.

The majority of the available processors when power is applied to them go to a default address location, and after finding the first bytes of binary data, start executing them. Based on this information, the hardware designers define the layout for the flash memory and the address ranges that could later be used to load and boot the Linux operating system from predictable addresses.

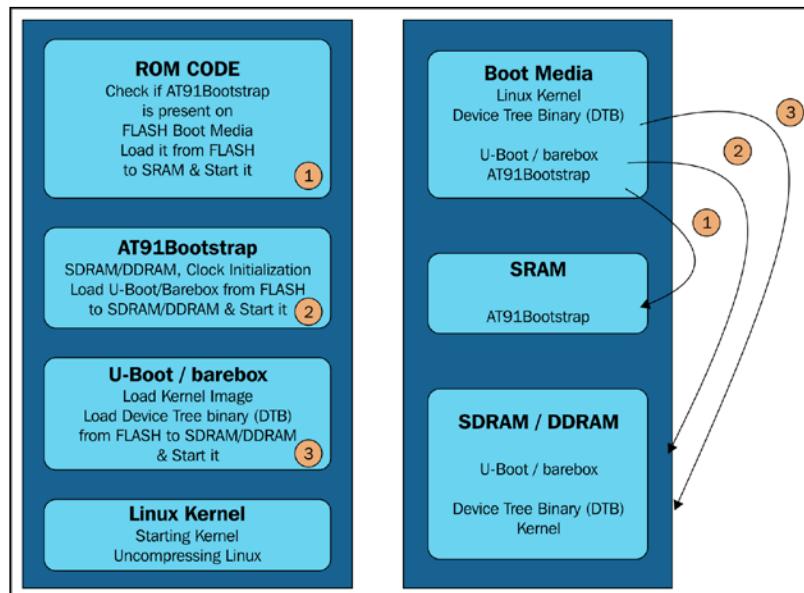
In the first stage of initialization, the board init is done, usually in the assembler language specific to the processor and after this is finished, the entire ecosystem is prepared for the operating system booting process. The bootloader is responsible for this; it is the component that offers the possibility to load, locate, and execute primary components of the operating system. Additionally, it can contain other advanced features, such as the capability to upgrade the OS image, validate an OS image, choose between several OS images, and even the possibility to upgrade itself. The difference between the traditional PC BIOS and an embedded bootloader is the fact that in an embedded environment, the bootloader is overwritten after the Linux kernel starts execution. It, in fact, ceases to exist after it offers control to the OS image.

Bootloaders need to carefully initialize peripherals, such as flash or DRAM, before they are used. This is not an easy task to do. For example, the DRAM chips cannot be read or written in a direct method - each chip has a controller that needs to be enabled for read and write operations. At the same time, the DRAM needs to be continually refreshed because the data will be lost otherwise. The refresh operation, in fact, represents the reading of each DRAM location within the time frame mentioned by the hardware manufacturer. All these operations are the DRAM controller's responsibility, and it can generate a lot of frustration for the embedded developer because it requires specific knowledge about the architecture design and DRAM chip.

A bootloader does not have the infrastructure that a normal application has. It does not have the possibility to only be called by its name and start executing. After being switched on when it gains control, it creates its own context by initializing the processor and necessary hardware, such as DRAM, moves itself in the DRAM for faster execution, if necessary and finally, starts the actual execution of code.

The first element that poses as a complexity is the compatibility of the start up code with the processor's boot sequence. The first executable instructions need to be at a predefined location in the flash memory, which is dependent of the processor and even hardware architecture. There is also the possibility for a number of processors to seek for those first executable instructions in several locations based on the hardware signals that are received.

Another possibility is to have the same structure on many of the newly available development boards, such as the Atmel SAMA5D3-Xplained:



For the Atmel SAMA5D3-Xplained board and others similar to it, the booting starts from an integrated boot code available in the ROM memory called BootROM on AT91 CPUs, which loads the first stage bootloader called AT91Bootstrap on SRAM and starts it. The first stage bootloader initializes the DRAM memory and starts the second stage bootloader, which is U-Boot in this case. More information on boot sequence possibilities can be found in the boot sequence header available, which you'll read about shortly.

The lack of an execution context represents another complexity. Having to write even a simple "Hello World" in a system without a memory and, therefore, without a stack on which to allocate information, would look very different from the well-known "Hello World" example. This is the reason why the bootloader initializes the RAM memory to have a stack available and is able to run higher-level programs or languages, such as C.

Comparing various bootloaders

As we read earlier, a number of bootloaders are available for embedded systems. The ones that will be presented here are as follows:

- **U-Boot:** This is also called the Universal Bootloader and is available mostly for PowerPC and ARM architectures for embedded Linux systems
- **Barebox:** This was initially known as U-Boot v2 and was started in 2007 with the scope to solve the limitations of U-Boot; it changed its name over time because the design goals and community changed
- **RedBoot:** This is a RedHat bootloader derived from eCos, an open-source real-time operating system that is portable and devised for embedded systems
- **rrload:** This is a bootloader for ARM and is based on embedded Linux systems
- **PPCBOOT:** This is a bootloader for PowerPC and is based on embedded Linux systems
- **CLR/OHH:** This represents a flash bootloader for embedded Linux systems based on an ARM architecture
- **Alios:** This is a bootloader that is written mostly in assembler, does ROM and RAM initializations, and tries to completely remove the need for firmware on embedded systems

There are a number of bootloaders available and this is a natural outcome of the fact that there are a huge number of different architectures and devices, so many, in fact, that it is almost near impossible to have one that would be good for all systems. The variety of bootloaders is high; the differentiator factors are represented by the board types and structure, SOC differences and even CPUs.

Delving into the bootloader cycle

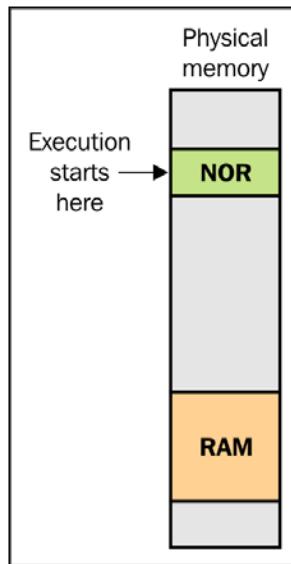
As mentioned previously, the bootloader is the component that is first run after initializing the system, and prepares the entire ecosystem for the operating system boot process. This process differs from one architecture to the other. For example, for the x86 architecture, the processor has access to BIOS, a piece of software available in a nonvolatile memory, which is usually a ROM. Its role starts out after resetting the system when it is executed and initializes the hardware components that will later be used by the first stage bootloader. It also executes the first stage of the bootloader.

The first stage bootloader is very small in terms of dimensions - in general, it is only 512 bytes and resides on a volatile memory. It performs the initialization for the full bootloader during the second stage. The second stage bootloaders usually reside next to the first stage ones, they contain the most number of features and do most of the work. They also know how to interpret various filesystem formats, mostly because the kernel is loaded from a filesystem.

For x86 processors, there are more bootloader solutions that are available:

- **GRUB:** The Grand Unified Bootloader is the most used and powerful bootloader available for Linux systems from desktop PC platforms. It is a component of the GNU Project and is one of the most potent bootloaders available for x86 architecture systems. This is because it is able to understand a large variety of filesystems and kernel images formats. It is able to change the boot configuration during boot time. GRUB also has support for a network boot and command-line interface. It has a configuration file that is processed at boot time and can be modified. More information about it can be found at <http://www.gnu.org/software/grub/>.
- **Lilo:** The Linux Loader a bootloader mostly used in commercial Linux distributions. Similar to the previous point, it is available for desktop PC platforms. It has more than one component, the first component for historical reasons is available on the first sector of a disk drive; it is the bootstrap component. Due to the same historical reasons, it is limited to the 512 bytes dimension and it loads and offers control to the second stage bootloader that does most of the bootloader's work. Lilo has a configuration utility that is mainly used as a source of information for the Linux kernel booting process. More information about it can be found at <http://www.tldp.org/HOWTO/LILO.html>.
- **Syslinux:** It is used for removable media or network booting. Syslinux is a Linux operating system bootloader that runs on MS-DOS or Windows FAT filesystems and is mainly used for rescue and first time installations of Linux. More information on it can be found at <http://www.kernel.org/pub/linux/utils/boot/syslinux/>.

For most embedded systems, this booting process does not apply, although there are some that replicate this behavior. There are two types of situations that will be presented next. The first one is a situation where the code execution starts from a fixed address location, and the second one refers to a situation where the CPU has a code available in the ROM memory that is called.



The right-hand side of the image is presented as the previously mentioned booting mechanism. In this case, the hardware requires a NOR flash memory chip, available at the start address to assure the start of the code execution.

A NOR memory is preferred over the NAND one because it allows random address access. It is the place where the first stage bootloader is programmed to start the execution, and this doesn't make it the most practical mechanism of booting.

Although it is not the most practical method used for the bootloader boot process, it is still available. However, it somehow becomes usable only on boards that are not suitable for more potent booting options.

The U-Boot bootloader

There are many open source bootloaders available today. Almost all of them have features to load and execute a program, which usually involves the operating system, and its features are used for serial interface communication. However, not all of them have the possibility to communicate over Ethernet or update themselves. Another important factor is represented by the widespread use of the bootloader. It is very common for organizations and companies to choose only one bootloader for the diversity of boards, processors, and architectures that they support. A similar thing happened with the Yocto Project when a bootloader was chosen to represent the official supported bootloader. They, and other similar companies, chose U-Boot bootloader, which is quite well known in the Linux community.

The U-Boot bootloader, or Das U-Boot as its official name, is developed and maintained by Wolfgang Denx with the support of the community behind it. It is licensed under GPLv2, its source code is freely available inside a git repository, as shown in the first chapter, and it has a two month intervals between releases. The release version name is shown as `U-boot vYYYY.MM`. The information about U-Boot loader is available at <http://www.denx.de/wiki/U-Boot/ReleaseCycle>.

The U-Boot source code has a very well defined directory structure. This can be easily seen with this console command:

```
tree -d -L 1
.
├── api
├── arch
├── board
├── common
├── configs
├── disk
├── doc
├── drivers
└── dts
```

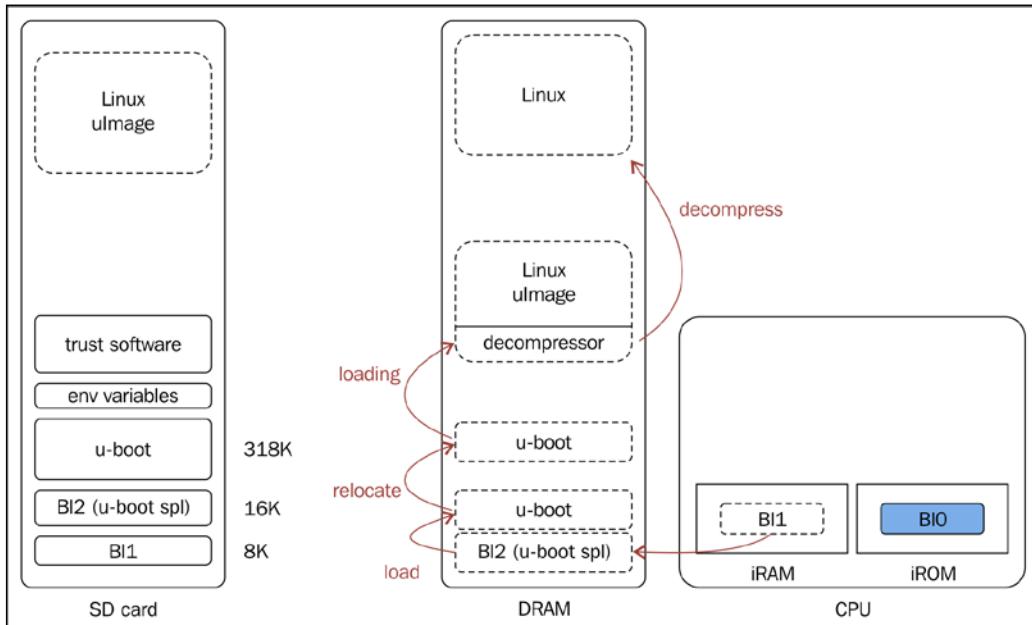
```
|── examples
|── fs
|── include
|── lib
|── Licenses
|── net
|── post
|── scripts
|── test
└── tools
19 directories
```

The arch directory contains architecture-specific files and directories-specific to each architecture, CPU or development board. An api contains external applications that are independent of a machine or architecture type. A board contains inside boards with specific names of directories for all board-specific files. A common is a place where misc functions are located. A disk contains disk drive handling functions, and documentation is available inside the doc directory. Drivers are available in the drivers directory. The filesystem-specific functionality is available inside the fs directory. There are still some directories that would need mentioning here, such as the include directory, which contains the header files; the lib directory contains generic libraries with support for various utilities, such as the flatten device tree, various decompressions, a post (Power On Self-Test) and others, but I will let them be discovered by the reader's curiosity, one small hint would be to inspect the README file in the *Directory Hierachy* section.

Moving through the U-Boot sources, which were downloaded in the previous chapter inside the ./include/configs file, configuration files can be found for each supported board. These configuration file is an .h file that contains a number of CONFIG_ files and defines information on memory mapping, peripherals and their setup, command line output, such as the boot default addresses used for booting a Linux system, and so on. More information on the configuration files could be found inside the README file in the *Configuration Options*, section or in a board specific configuration file. For Atmel SAMA5D3-Xplained, the configuration file is include/configs/sama5d3_xplained.h. Also, there are two configurations available for this board in the configs directory, which are as follows:

- configs/sama5d3_xplained_mmc_defconfig
- configs/sama5d3_xplained_nandflash_defconfig

These configurations are used to define the board **Secondary Program Loader (SPL)** initialization method. SPL represents a small binary built from the U-Boot source code that is placed on the SRAM memory and is used to load the U-Boot into the RAM memory. Usually, it has less than 4 KB of memory, and this is how the booting sequence looks:



Before actually starting the build for the U-Boot source code for a specific board, the board configuration must be specified. For the Atmel SAMA5_Xplained development board, as presented in the preceding image, there are two available configurations that could be done. The configuration is done with the make command:

```
ARCH=arm CROSS_COMPILE=${CC} sama5d3_xplained_nandflash_defconfig
```

Behind this command, the include/config.h file is created. This header include definitions that are specific for the chosen board, architecture, CPU, and also board-specific header includes. The defined CONFIG_* variable read from the include/config.h file includes determining the compilation process. After the configuration is completed, the build can be started for the U-Boot.

Another example that can be very useful when inspected relates to the other scenario of booting an embedded system, one that requires the use of a NOR memory. In this situation, we can take a look at a particular example. This is also well described inside the *Embedded Linux Primer* by Christopher Hallinan, where a processor of the AMCC PowerPC 405GP is discussed. The hardcoded address for this processor is 0xFFFFFFFFFC and is visible using `.resetvec`, the reset vector placement. There also specifies the fact that the rest of this section is completed with only the value 1 until the end of the 0xFFFFFFFF stack; this implies that an empty flash memory array is completed only with values of 1. The information about this section is available in `resetvec.s` file, which is located at `arch/powerpc/cpu/ppc4xx/resetvec.s`. The contents of `resetvec.s` file is as follows:

```
/* Copyright MontaVista Software Incorporated, 2000 */
#include <config.h>
.section .resetvec, "ax"
#if defined(CONFIG_440)
    b _start_440
#else
    #if defined(CONFIG_BOOT_PCI) && defined(CONFIG_MIP405)
        b _start_pci
    #else
        b _start
    #endif
#endif
```

On inspection of this file's source code, it can be seen that only an instruction is defined in this section independently of the available configuration options.

The configuration for the U-Boot is done through two types of configuration variables. The first one is `CONFIG_*`, and it makes references to configuration options that can be configured by a user to enable various operational features. The other option is called `CFG_*` and this is used for configuration settings and to make references to hardware-specific details. The `CFG_*` variable usually requires good knowledge of a hardware platform, peripherals and processors in general. The configure file for the SAMA5D3 Xplained hardware platform is available inside the `include/config.h` header file, as follows:

```
/* Automatically generated - do not edit */
#define CONFIG_SAMA5D3 1
#define CONFIG_SYS_USE_NANDFLASH 1
#define CONFIG_SYS_ARCH "arm"
#define CONFIG_SYS_CPU "armv7"
#define CONFIG_SYS_BOARD "sama5d3_xplained"
#define CONFIG_SYS_VENDOR "atmel"
#define CONFIG_SYS_SOC "at91"
#define CONFIG_BOARDDIR board/atmel/sama5d3_xplained
#include <config_cmd_defaults.h>
```

```
#include <config_defaults.h>
#include <configs/sama5d3_xplained.h>
#include <asm/config.h>
#include <config_fallbacks.h>
#include <config_uncmd_spl.h>
```

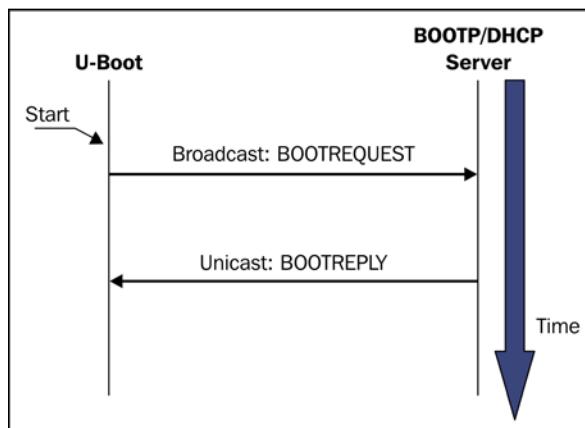
The configuration variables available here represent the corresponding configurations for the SAMA5D3 Xplained board. A part of these configurations refer to a number of standard commands available for user interactions with the bootloader. These commands can be added or removed for the purpose of extending or subtracting commands from the available command line interface.

More information on the U-Boot configurable command interface can be found at <http://www.denx.de/wiki/view/DULG/UBootCommandLineInterface>.

Booting the U-Boot options

In an industrial environment, interaction with the U-Boot is mainly done through the Ethernet interface. Not only does an Ethernet interface enable the faster transfer of operating system images, but it is also less prone to errors than a serial connection.

One of the most important features available inside a bootloader is related to the support for **Dynamic Host Control Protocol (DHCP)**, **Trivial File Transfer Protocol (TFTP)**, and even **Bootstrap Protocol (BOOTP)**. BOOTP and DHCP enable an Ethernet connection to configure itself and acquire an IP address from a specialized server. TFTP enables the download of files through a TFTP server. The messages passed between a target device and the DHCP/BOOTP servers are represented in the following image in a more generic manner. Initially, the hardware platform sends a broadcast message that arrives at all the DHCP/BOOTP servers available. Each server sends back its offer, which also contains an IP address, and the client accepts the one that suits its purposes the best and declines the other ones.



After the target device has finished communication with DHCP/BOOTP, it remains with a configuration that is specific to the target and contains information, such as the hostname, target IP and hardware Ethernet address (MAC address), netmask, tftp server IP address and even a TFTP filename. This information is bound to the Ethernet port and is used later in the booting process.

To boot images, U-Boot offers a number of capabilities that refer to the support of storage subsystems. These options include the RAM boot, MMC boot, NAND boot, NFS boot and so on. The support for these options is not always easy and could imply both hardware and software complexity.

Porting U-Boot

I've mentioned previously that U-Boot is one of the most used and known bootloaders available. This is also due to the fact that its architecture enables the porting of new development platforms and processors in a very easy manner. At the same time, there are a huge number of development platforms available that could be used as references. The first thing that any developer who is interested in porting a new platform should do is to inspect the board and arch directories to establish their baselines, and, at the same time, also identify their similarities with other CPUs and available boards.

The `board.cfg` file is the starting point to register a new platform. Here, the following information should be added as a table line:

- Status
- Architecture
- CPU
- SOC
- Vendor
- Board name
- Target
- Options
- Maintainers

To port a machine similar to SAMA5D3 Xplained, one of the directories that could be consulted is the `arch` directory. It contains files, such as `board.c`, with information related to the initialization process for boards and SOCs. The most notable processes are `board_init_r()`, which does the setup and probing for board and peripherals after its relocation in the RAM, `board_init_f()`, which identifies the stack size and reserved address before its relocation in the RAM, and `init_sequence[]`, which is called inside the `board_init_f` for the setup of peripherals. Other important files inside the same locations are the `bootm.c` and `interrupts.c` files. The former has the main responsibility of the boot from memory of the operating system, and the latter is responsible for implementation of generic interrupts.

The board directory also has some interesting files and functions that need to be mentioned here, such as the `board/atmel/sama5d3_xplained/sama5d3_xplained.c` file. It contains functions, such as `board_init()`, `dram_init()`, `board_eth_init()`, `board_mmc_init`, `spl_board_init()`, and `mem_init()` that are used for initialization, and some of them called by the `arch/arm/lib/board.c` file.

Here are some other relevant directories:

- `common`: This holds information about user commands, middleware, APIs that perform the interfacing between the middleware and user commands, and other functions and functionalities used by all available boards.
- `drivers`: This contains drivers for various device drivers and middleware APIs, such as `drivers/mmc/mmc.c`, `drivers/pci/pci.c`, `drivers/watchdog/at91sam9_wdt.c` and so on.
- `fs`: Various supported filesystems, such as USB, SD Card, Ext2 FAT, and so on are available here.
- `include`: This represents the location where all the headers necessary for most of the boards are present. SOCs and other software is also available. Inside `include/configs`, board-specific configurations are available, and include the headers imported from Linux; these could be used for various device drivers, porting, or other byte operations.
- `tools`: This is the place where tools, such as `checkpatch.pl`, a patch examination tool used as a coding style check, are used before sending it to the mailing list or the `mkimage.c` tool. This is also used for the U-Boot generic header generation that makes Linux binaries, and assures that they are able to be booted using U-Boot.

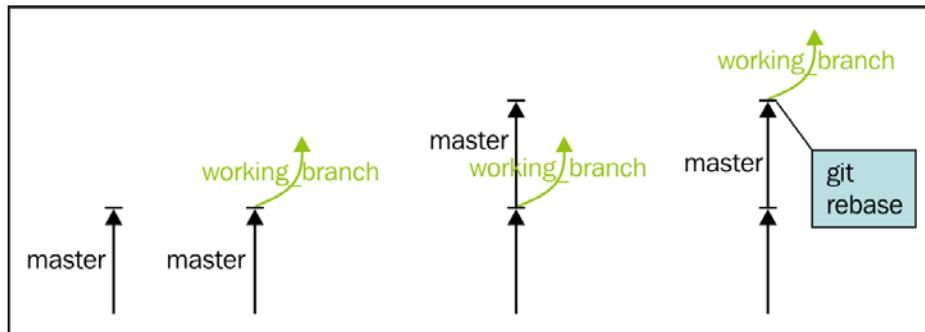
More information about the SAMA5D3 Xplained board can be found by inspecting the corresponding doc directory and README files, such as README.at91, README.at91-soc, README.atmel_mci, README.atmel_pmecc, README.ARM-memory-map, and so on.

For people interested in committing to the changes they made while porting a new development board, CPU, or SOC to U-Boot, a few rules should be followed. All of these are related to the git interaction and help you to ensure the proper maintenance of your branches.

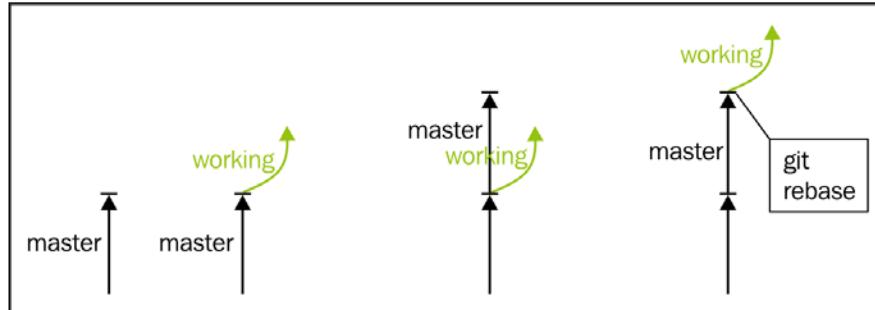
The first thing that a developer should do is to track the upstream branch that corresponds to a local branch. Another piece of advice would be to forget about `git merge` and instead use `git rebase`. Keeping in contact with the upstream repository can be done using the `git fetch` command. To work with patches, some general rules need to be followed, and patches need to have only one logical change, which can be any one of these:

- Changes should not contain unrelated or different modifications; only one patch is available and acceptable per changeset
- Commits should make the use of `git-bisect` where possible while detecting bugs in sources, when necessary
- If multiple files are impacted by a set of modifications, all of them should be submitted in the same patch
- Patches need to have review, and a very thorough one at that

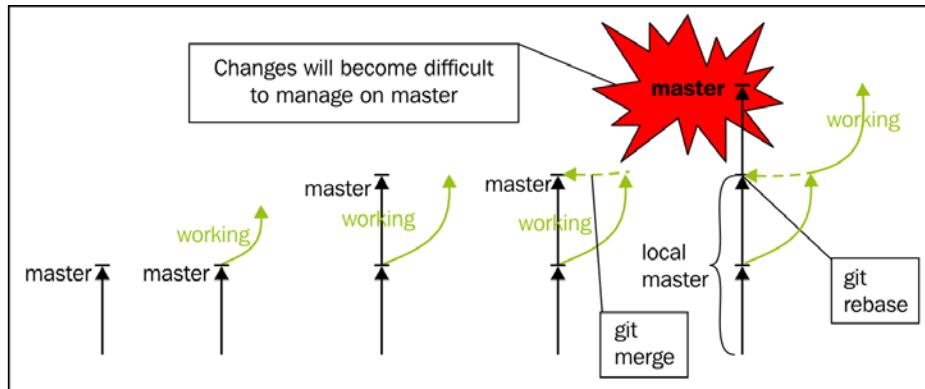
Let's take a look at following diagram, which illustrates the git rebase operation:



As shown in both the preceding and following diagram, the **git rebase** operation has recreated the work from one branch onto another. Every commit from one branch is made available on the succeeding one, just after the last commit from it.



The **git merge** operation, on the other hand, is a new commit that has two parents: the branch from which it was ported, and the new branch on which it was merged. In fact, it gathers a series of commits into one branch with a different commit ID, which is why they are difficult to manage.



More information related to git interactions can be found at <http://git-scm.com/documentation> or <http://www.denx.de/wiki/U-Boot/Patches>.

Almost always when porting a new feature in U-Boot, debugging is involved. For a U-Boot debugger, there are two different situations that can occur:

- The first situation is when `lowlevel_init` was not executed
- The second situation is when the `lowlevel_init` was executed; this is the most well known scenario

In the next few lines, the second situation will be considered: the baseline enabling a debugging session for U-Boot. To make sure that debugging is possible, the `elf` file needs to be executed. Also, it cannot be manipulated directly because the linking address will be relocated. For this, a few tricks should be used:

- The first step is to make sure that the environment is clean and that old objects are not available any more: `make clean`
- The next step would be to make sure the dependencies are cleaned:
`find ./ | grep depend | xargs rm`
- After the cleaning is finished, the target build can start and the output can be redirected inside a log file: `make sama5d3_xplained 2>&1 > make.log`
- The generated output should be renamed to avoid debugging problems for multiple boards: `mv u-boot.bin u-boot_sama5d3_xplained.bin`
- It is important to enable DEBUG in the board configuration file; inside `include/configs/ sama5d3_xplained.h`, add the `#define DEBUG` line

An early development platform can be set up after relocation takes place and the proper breakpoint should be set after the relocation has ended. A symbol needs to be reloaded for U-Boot because the relocation will move the linking address. For all of these tasks, a `gdb` script is indicated as `gdb_gdb-script.sh`:

```
#!/bin/sh
${CROSS_COMPILE}-gdb u-boot -command=gdb-command-script.txt

vim gdb-command-script.txt
target remote ${ip}:${port}
load
set symbol-reloading
# break the process before calling board_init_r() function
b start.S:79
c
...
# the symbol-file need to be align to the address available after
relocation
add-symbol-file u-boot ${addr}
# break the process at board_init_r() function for single stepping
b board.c:494
```



More information on relocation can be found at doc/README.
arm-relocation.



The Yocto Project

The Yocto Project uses various recipes to define interactions to each of the supported bootloaders. Since there are multiple stages of booting, there are also multiple recipes and packages required inside the BSP. The recipes available for various bootloaders are not different from any other recipes available in the Yocto world. However, they have some details that make them unique.

The board that we will focus on here is the sama5d3_xplained development board, and it is available inside the `meta-atmel` layer. Inside this layer, the corresponding recipes for the first and second stage bootloaders can be found inside the `recipes-bsp` directory. Here, I am referring to the `at91bootstrap` and `u-boot` recipes. There are some misconceptions about first stage and second stage bootloaders. They might be referred to as second level and third level bootloaders, because the boot ROM code may or may not be taken into account during a discussion. In this book, we prefer to call them as first stage and second stage bootloaders.

The `AT91bootstrap` package represents the first-stage bootloader from Atmel available for their SOCs. It manages hardware initialization and also executes the second stage bootloader download from a boot media inside the memory; it starts it at the end. In the `meta-atmel` layer, the second stage bootloader is `u-boot`, and it is later used for the Linux operating system boot.

Usually, inside a BSP layer, the support for multiple development boards is offered, and this means that multiple versions and bootloader packages are offered as well. The distinction between them, however, is on the basis of machine configurations. For the SAMA5D3 Xplained development board, the machine configuration is available inside the `conf/machine/sama5d3_xplained` file. In this file, the preferred bootloader versions, providers, and configurations are defined. If these configurations are not MACHINE specific, they could very well be performed inside the package recipe.

This is one example of the configurations available for the `sama5d3_xplained` development board:

```
PREFERRED_PROVIDER_virtual/bootloader = "u-boot-at91"
UBOOT_MACHINE ?= "sama5d3_xplained_nandflash_config"
UBOOT_ENTRYPOINT = "0x20008000"
UBOOT_LOADADDRESS = "0x20008000"

AT91BOOTSTRAP_MACHINE ?= "sama5d3_xplained"
```

Summary

In this chapter, you were presented with information on bootloaders, with particular focus on the U-Boot bootloader. We also discussed topics related to U-Boot interaction, porting, debugging, general information on bootloaders, U-Boot alternatives and a boot sequence inside an embedded environment. There was also a section related to the Yocto Project, where you were introduced to the mechanism used to support various bootloaders available inside BSP. A number of exercises were presented across the chapter, and they offered more clarity on this subject.

In the next chapter, we will discuss the Linux kernel, its features and source code, modules and drivers, and, in general, most of the information needed to interact with the Linux kernel. As you have already been introduced to it, we will also concentrate on the Yocto Project and how it is able to work with various kernel versions for a number of boards and exercises. This should ease the understanding of the information presented to you.

4

Linux Kernel

In this chapter, you will not only learn about the Linux kernel in general, but also specific things about it. The chapter will start with a quick presentation of the history of Linux and its role and will then continue with an explanation of its various features. The steps used to interact with the sources of the Linux kernel will not be omitted. You will only be presented with the steps necessary to obtain a Linux kernel image from a source code, but also information about what porting for a new **ARM machine** implies, and some of the methods used to debug various problems that could appear when working with the Linux kernel sources in general. In the end, the context will be switched to the Yocto Project to show how the Linux kernel can be built for a given machine, and also how an external module can be integrated and used later from a root filesystem image.

This chapter will give you an idea of the Linux kernel and Linux operating system. This presentation would not have been possible without the historical component. Linux and UNIX are usually placed in the same historical context, but although the Linux kernel appeared in 1991 and the Linux operating system quickly became an alternative to the UNIX operating system, these two operating systems are members of the same family. Taking this into consideration, the history of UNIX operating system could not have started from another place. This means that we need to go back in time to more than 40 years ago, to be more precise, about 45 years ago to 1969 when Dennis Ritchie and Ken Thompson started the development of UNIX.

The predecessor of UNIX was **Multiplexed Information and Computing Service (Multics)**, a multiuser operating system project that was not on its best shape at the time. Since the Multics had become a nonviable solution for Bell Laboratories Computer Sciences Research Center in the summer of 1969, a filesystem design was born and it later became what is known today as UNIX. Over time, it was ported on multiple machines due to its design and the fact that the source code was distributed alongside it. The most prolific contributor to the UNIX was the University of California, Berkeley. They also developed their own UNIX version called **Berkeley Software Distribution (BSD)**, that was first released in 1977. Until the 1990s, multiple companies developed and offered their own distributions of UNIX, their main inspirations being Berkeley or AT&T. All of them helped UNIX become a stable, robust, and powerful operating system. Among the features that made UNIX strong as an operating system, the following can be mentioned:

- UNIX is simple. The number of system calls that it uses are reduced to only a couple of hundred and their design is basic
- Everything is regarded as a file in UNIX, making the manipulation of data and devices simpler, and it minimizes system calls used for interaction.
- Faster process creation time and the `fork()` system call.
- The UNIX kernel and utilities written in C language as well as a property that makes it easily portable and accessible.
- Simple and robust **interprocess communication (IPC)** primitives helps in the creation of fast and simple programs that accomplish only one thing in the best available manner.

Nowadays, UNIX is a mature operating system with support for features, such as virtual memory, TCP/IP networking, demand paging preemptive multiprocessing, and multithreading. The features spread is wide and varies from small embedded devices to systems with hundreds of processors. Its development has moved past the idea that UNIX is a research project, and it has become an operating system that is general-purpose and practically fits any needs. All this has happened due to its elegant design and proven simplicity. It was able to evolve without losing its capability to remain simple.

Linux is as an alternative solution to a UNIX variant called **Minix**, an operating system that was created for teaching purposes, but it lacked easy interaction with the system source code. Any changes made to the source code were not easily integrated and distributed because of Minix's license. Linus Torvalds first started working at a terminal emulator to connect to other UNIX systems from his university. Within the same academic year, emulator evolved in a full-fledged UNIX. He released it to be used by everyone in 1991.

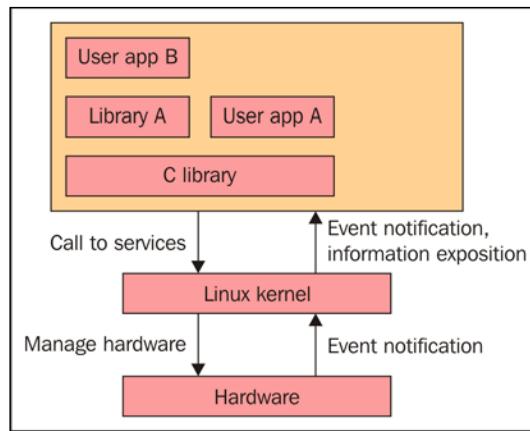
One of the most attractive features of Linux is that it is an open source operating system whose source code is available under the GNU GPL license. When writing the Linux kernel, Linus Torvalds used the best design choices and features from the UNIX available in variations of the operating system kernel as a source of inspiration. Its license is what has propelled it into becoming the powerhouse it is today. It has engaged a large number of developers that helped with code enhancements, bug fixing, and much more.

Today, Linux is an experienced operating system that is able to run on a multitude of architectures. It is able to run on devices that are even smaller than a wristwatch or on clusters of supercomputer. It's the new sensation of our days and is being adopted by companies and developers around the world in an increasingly diversified manner. The interest in the Linux operating system is very strong and this implies not only diversity, but also offers a great number of benefits, ranging from security, new features, embedded solutions to server solution options, and many more.

Linux has become a truly collaborative project developed by a huge community over the internet. Although a great number of changes were made inside this project, Linus has remained its creator and maintainer. Change is a constant factor in everything around us and this applies to Linux and its maintainer, who is now called Greg Kroah-Hartman, and has already been its kernel maintainer for two years now. It may seem that in the period that Linus was around, the Linux kernel was a loose-knit community of developers. This may be because of Linus' harsh comments that are known worldwide. Since Greg has been appointed the kernel maintainer, this image started fading gradually. I am looking forward to the years to come.

The role of the Linux kernel

With an impressive numbers of code lines, the Linux kernel is one of the most prominent open source projects and at the same time, the largest available one. The Linux kernel constitutes a piece of software that helps with the interfacing of hardware, being the lowest-level code available that runs in everyone's Linux operating system. It is used as an interface for other user space applications, as described in the following diagram:



The main roles of the Linux kernel are as follows:

- It provides a set of portable hardware and architecture APIs that offer user space applications the possibility to use necessary hardware resources
- It helps with the management of hardware resources, such as a CPU, input/output peripherals, and memory
- It is used for the management of concurrent accesses and the usage of necessary hardware resources by different applications.

To make sure that the preceding roles are well understood, an example will be very useful. Let's consider that in a given Linux operating system, a number of applications need access to the same resource, a network interface, or a device. For these elements, the kernel needs to multiplex a resource in order to make sure that all applications have access to it.

Delving into the features of the Linux kernel

This section will introduce a number of features available inside the Linux kernel. It will also cover information about each of them, how they are used, what they represent, and any other relevant information regarding each specific functionality. The presentation of each feature familiarizes you with the main role of some of the features available inside the Linux kernel, as well as the Linux kernel and its source code in general.

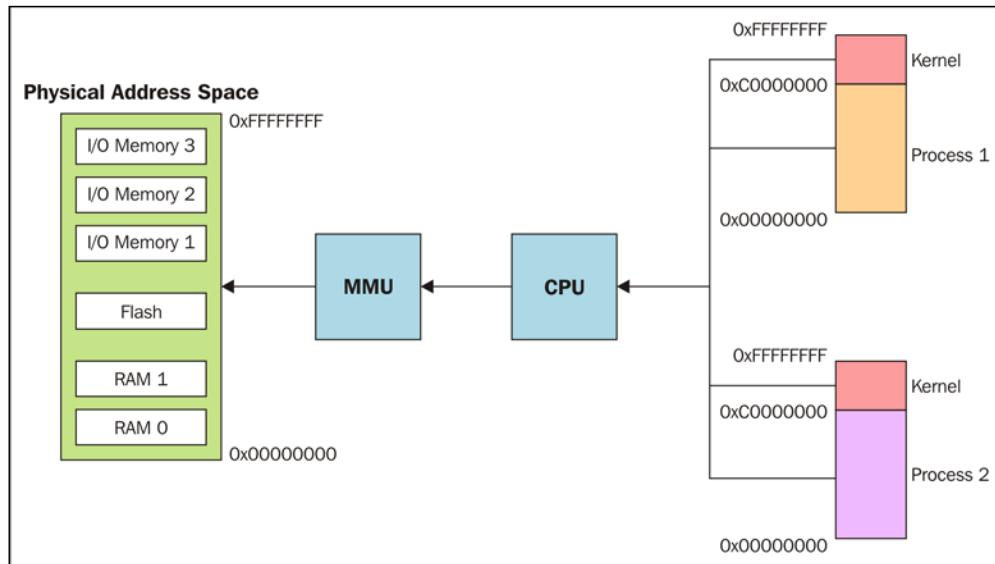
On a more general note, some of the most valuable features that the Linux kernel has are as follows:

- Stability and reliability
- Scalability
- Portability and hardware support
- Compliance with standards
- Interoperability between various standards
- Modularity
- Ease of programming
- Comprehensive support from the community
- Security

The preceding features does not constitute actual functionalities, but have helped the project along its development process and are still helping it today. Having said this, there are a lot of features that are implemented, such as fast user space mutex (futex), netfilterers, Simplified Mandatory Access Control Kernel (smack), and so on. A complete list of these can be accessed and studied at http://en.wikipedia.org/wiki/Category:Linux_kernel_features.

Memory mapping and management

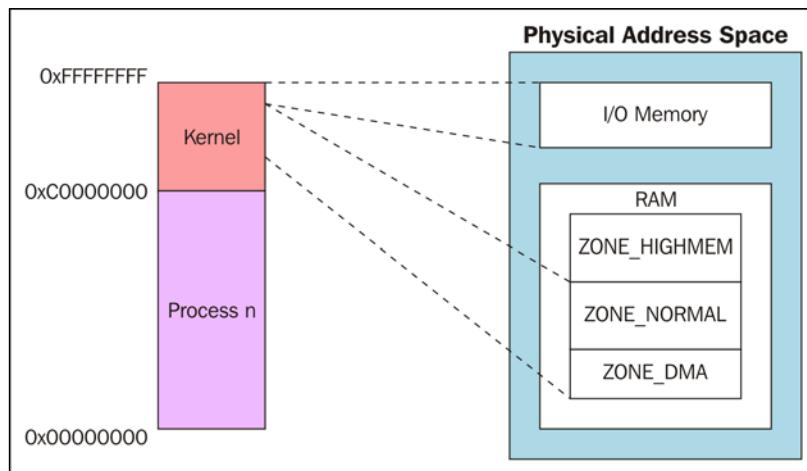
When discussing the memory in Linux, we can refer to it as the physical and virtual memory. Compartments of the RAM memory are used for the containment of the Linux kernel variables and data structures, the rest of the memory being used for dynamic allocations, as described here:



The physical memory defines algorithms and data structures that are able to maintain the memory, and it is done at the page level relatively independently by the virtual memory. Here, each physical page has a `struct page` descriptor associated with it that is used to incorporate information about the physical page. Each page has a `struct page` descriptor defined. Some of the fields of this structure are as follows:

- `_count`: This represents the page counter. When it reaches the 0 value, the page is added to the free pages list.
- `virtual`: This represents the virtual address associated to a physical page. The `ZONE_DMA` and `ZONE_NORMAL` pages are always mapped, while the `ZONE_HIGHMEM` are not always mapped.
- `flags`: This represents a set of flags that describe the attributes of the page.

The zones of the physical memory have been previously. The physical memory is split up into multiple nodes that have a common physical address space and a fast local memory access. The smallest of them is **ZONE_DMA** between 0 to 16Mb. The next is **ZONE_NORMAL**, which is the LowMem area between 16Mb to 896Mb, and the largest one is **ZONE_HIGHMEM**, which is between 900Mb to 4GB/64Gb. This information can be visible both in the preceding and following images:



The virtual memory is used both in the user space and the kernel space. The allocation for a memory zone implies the allocation of a physical page as well as the allocation of an address space area; this is done both in the page table and in the internal structures available inside the operating system. The usage of the page table differs from one architecture type to another. For the **Complex instruction set computing (CISC)** architecture, the page table is used by the processor, but on a **Reduced instruction set computing (RISC)** architecture, the page table is used by the core for a page lookup and **translation lookaside buffer (TLB)** add operations. Each zone descriptor is used for zone mapping. It specifies whether the zone is mapped for usage by a file if the zone is read-only, copy-on-write, and so on. The address space descriptor is used by the operating system to maintain high-level information.

The memory allocation is different between the user space and kernel space context because the kernel space memory allocation is not able to allocate memory in an easy manner. This difference is mostly due to the fact that error management in the kernel context is not easily done, or at least not in the same key as the user space context. This is one of the problems that will be presented in this section along with the solutions because it helps readers understand how memory management is done in the context of the Linux kernel.

The methods used by the kernel for memory handling is the first subject that will be discussed here. This is done to make sure that you understand the methods used by the kernel to obtain memory. Although the smallest addressable unit of a processor is a byte, the **Memory Management Unit (MMU)**, the unit responsible for virtual to physical translation the smallest addressable unit is the page. A page's size varies from one architecture to another. It is responsible for maintaining the system's page tables. Most of 32-bit architectures use 4KB pages, whereas the 64-bit ones usually have 8KB pages. For the Atmel SAMA5D3-Xplained board, the definition of the `struct page` structure is as follows:

```
struct page {  
    unsigned long    flags;  
    atomic_t         _count;  
    atomic_t         _mapcount;  
    struct address_space *mapping;  
    void            *virtual;  
    unsigned long   debug_flags;  
    void            *shadow;  
    int             _last_nid;  
  
};
```

This is one of the most important fields of the page structure. The `flags` field, for example, represents the status of the page; this holds information, such as whether the page is dirty or not, locked, or in another valid state. The values that are associated with this flag are defined inside the `include/linux/page-flags-layout.h` header file. The `virtual` field represents the virtual address associated with the page, `count` represents the count value for the page that is usually accessible indirectly through the `page_count()` function. All the other fields can be accessed inside the `include/linux/mm_types.h` header file.

The kernel divides the hardware into various zone of memory, mostly because there are pages in the physical memory that are not accessible for a number of the tasks. For example, there are hardware devices that can perform DMA. These actions are done by interacting with only a zone of the physical memory, simply called `ZONE_DMA`. It is accessible between 0-16 Mb for x86 architectures.

There are four main memory zones available and other two less notable ones that are defined inside the kernel sources in the `include/linux/mmzone.h` header file. The zone mapping is also architecture-dependent for the Atmel SAMA5D3-Xplained board. We have the following zones defined:

```
enum zone_type {  
    #ifdef CONFIG_ZONE_DMA
```

```
/*
 * ZONE_DMA is used when there are devices that are not able
 * to do DMA to all of addressable memory (ZONE_NORMAL). Then
we
 * carve out the portion of memory that is needed for these
devices.
 * The range is arch specific.
 *
 * Some examples
 *
 * Architecture      Limit
 * -----
 * parisc, ia64, sparc <4G
 * s390                <2G
 * arm                  Various
 * alpha                Unlimited or 0-16MB.
 *
 * i386, x86_64 and multiple other arches
 *                         <16M.
 */
ZONE_DMA,
#endif
#ifndef CONFIG_ZONE_DMA32
/*
 * x86_64 needs two ZONE_DMAs because it supports devices that
are
 * only able to do DMA to the lower 16M but also 32 bit
devices that
 * can only do DMA areas below 4G.
 */
ZONE_DMA32,
#endif
/*
 * Normal addressable memory is in ZONE_NORMAL. DMA operations
can be
 * performed on pages in ZONE_NORMAL if the DMA devices
support
 * transfers to all addressable memory.
 */
ZONE_NORMAL,
#ifndef CONFIG_HIGHMEM
/*
 * A memory area that is only addressable by the kernel
through
```

```
        * mapping portions into its own address space. This is for
example
        * used by i386 to allow the kernel to address the memory
beyond
        * 900MB. The kernel will set up special mappings (page
        * table entries on i386) for each page that the kernel needs
to
        * access.
        */
ZONE_HIGHMEM,
#endif
ZONE_MOVABLE,
__MAX_NR_ZONES
};
```

There are allocations that require interaction with more than one zone. One such example is a normal allocation that is able to use either `ZONE_DMA` or `ZONE_NORMAL`. `ZONE_NORMAL` is preferred because it does not interfere with direct memory accesses, though when the memory is at full usage, the kernel might use other available zones besides the ones that it uses in normal scenarios. The kernel that is available is a **struct zone** structure that defines each zone's relevant information. For the Atmel SAMA5D3-Xplained board, this structure is as shown here:

```
struct zone {
    unsigned long watermark[NR_WMARK];
    unsigned long percpu_drift_mark;
    unsigned long lowmem_reserve[MAX_NR_ZONES];
    unsigned long dirty_balance_reserve;
    struct per_cpu_pageset __percpu *pageset;
    spinlock_t lock;
    int all_unreclaimable;
    struct free_area free_area[MAX_ORDER];
    unsigned int compact_considered;
    unsigned int compact_defer_shift;
    int compact_order_failed;
    spinlock_t lru_lock;
    struct lruvec lruvec;
    unsigned long pages_scanned;
    unsigned long flags;
    unsigned int inactive_ratio;
    wait_queue_head_t *wait_table;
    unsigned long wait_table_hash_nr_entries;
    unsigned long wait_table_bits;
    struct pglist_data *zone_pgdat;
```

```
    unsigned long          zone_start_pfn;
    unsigned long          spanned_pages;
    unsigned long          present_pages;
    unsigned long          managed_pages;
    const char            *name;
};


```

As you can see, the zone that defines the structure is an impressive one. Some of the most interesting fields are represented by the `watermark` variable, which contain the high, medium, and low watermarks for the defined zone. The `present_pages` attribute represents the available pages within the zone. The `name` field represents the name of the zone, and others, such as the `lock` field, a spin lock that shields the zone structure for simultaneous access. All the other fields that can be identified inside the corresponding `include/linux/mmzone.h` header file for the Atmel SAMA5D3 Xplained board.

With this information available, we can move ahead and find out how the kernel implements memory allocation. All the available functions that are necessary for memory allocation and memory interaction in general, are inside the `linux/gfp.h` header file. Some of these functions are:

```
struct page * alloc_pages(gfp_t gfp_mask, unsigned int order)
```

This function is used to allocate physical pages in a continuous location. At the end, the return value is represented by the pointer of the first page structure if the allocation is successful, or `NULL` if errors occur:

```
void * page_address(struct page *page)
```

This function is used to get the logical address for a corresponding memory page:

```
unsigned long __get_free_pages(gfp_t gfp_mask, unsigned int order)
```

This one is similar to the `alloc_pages()` function, but the difference is that the return variable is offered in the `struct page * alloc_page(gfp_t gfp_mask)` return argument:

```
unsigned long __get_free_page(gfp_t gfp_mask)
struct page * alloc_page(gfp_t gfp_mask)
```

The preceding two functions are wrappers over similar ones, the difference is that this function returns only one page information. The order for this function has the zero value:

```
unsigned long get_zeroed_page(unsigned int gfp_mask)
```

The preceding function does what the name suggests. It returns the page full of zero values. The difference between this function and the `__get_free_page()` function is that after being released, the page is filled with zero values:

```
void __free_pages(struct page *page, unsigned int order)
void free_pages(unsigned long addr, unsigned int order)
void free_page(unsigned long addr)
```

The preceding functions are used for freeing the given allocated pages. The passing of the pages should be done with care because the kernel is not able to check the information it is provided.

Page cache and page writeback

Usually the disk is slower than the physical memory, so this is one of the reasons that memory is preferred over disk storage. The same applies for processor's cache levels: the closer it resides to the processor the faster it is for the I/O access. The process that moves data from the disk into the physical memory is called **page caching**. The inverse process is defined as **page writeback**. These two notions will be presented in this subsection, but is it mainly about the kernel context.

The first time the kernel calls the `read()` system call, the data is verified if it is present in the page cache. The process by which the page is found inside the RAM is called **cache hit**. If it is not available there, then data needs to be read from the disk and this process is called **cache miss**.

When the kernel issues the `write()` system call, there are multiple possibilities for cache interaction with regard to this system call. The easiest one is to not cache the write system calls operations and only keep the data in the disk. This scenario is called **no-write cache**. When the write operation updates the physical memory and the disk data at the same time, the operation is called **write-through cache**. The third option is represented by **write-back cache** where the page is marked as dirty. It is added to the dirty list and over time, it is put on the disk and marked as not dirty. The best synonym for the dirty keyword is represented by the synchronized key word.

The process address space

Besides its own physical memory, the kernel is also responsible for user space process and memory management. The memory allocated for each user space process is called **process address space** and it contains the virtual memory addressable by a given process. It also contains the related addresses used by the process in its interaction with the virtual memory.

Usually a process receives a flat 32 or 64-bit address space, its size being dependent on the architecture type. However, there are operating systems that allocate a **segmented address space**. The possibility of sharing the address space between the operating systems is offered to threads. Although a process can access a large memory space, it usually has permission to access only an interval of memory. This is called a **memory area** and it means that a process can only access a memory address situated inside a viable memory area. If it somehow tries to administrate a memory address outside of its valid memory area, the kernel will kill the process with the *Segmentation fault* notification.

A memory area contains the following:

- The `text` section maps source code
- The `data` section maps initialized global variables
- The `bss` section maps uninitialized global variables
- The `zero page` section is used to process user space stack
- The shared libraries `text`, `bss` and data-specific sections
- Mapped files
- Anonymous memory mapping is usually linked with functions, such as `malloc()`
- Shared memory segments

A process address space is defined inside the Linux kernel source through a **memory descriptor**. This structure is called `struct mm_struct`, which is defined inside the `include/linux/mm_types.h` header file and contains information relevant for a process address space, such as the number of processes that use the address space, a list of memory areas, the last memory area that was used, the number of memory areas available, start and finish addresses for the code, data, heap and stack sections.

For a kernel thread, no process address space associated with it; for kernel, the process descriptor structure is defined as `NULL`. In this way, the kernel mentions that a kernel thread does not have a user context. A kernel thread only has access to the same memory as all the other processes. A kernel thread does not have any pages in a user space or access to the user space memory.

Since the processors work only with physical addresses, the translation between physical and virtual memory needs to be made. These operations are done by the page tables that split the virtual addresses into smaller components with associated indexes that are used for pointing purposes. In the majority of available boards and architectures in general, the page table lookup is handled by the hardware; the kernel is responsible for setting it up.

Process management

A process, as presented previously, is a fundamental unit in a Linux operating system and at the same time, is a form of abstraction. It is, in fact, a program in execution, but a program by itself is not a process. It needs to be in an active state and have associated resources. A process is able to become a parent by using the `fork()` function, which spawns a child process. Both parent and child processes reside in separate address spaces, but both of them have the same content. The `exec()` family of function is the one that is able to execute a different program, create an address space, and load it inside that address space.

When `fork()` is used, the resources that the parent process has are reproduced for the child. This function is implemented in a very interesting manner; it uses the `clone()` system call that, at its base, contains the `copy_process()` function. This function does the following:

- Calls the `dup_task_struct()` function to create a new kernel stack. The `task_struct` and `thread_info` structures are created for a new process.
- Checks that the child does not go beyond the limits of the memory area.
- The child process distinguishes itself from its parent.
- It is set as `TASK_UNINTERRUPTIBLE` to make sure it does not run.
- Flags are updated.
- `PID` is associated with the child process.
- The flags that are already set are inspected and proper action is performed with respect to their values.
- The clean process is performed at the end when the child process pointer is obtained.

Threads in Linux are very similar to processes. They are viewed as processes that share various resources, such as memory address space, open files, and so on. The creation of threads is similar to a normal task, the exception being the `clone()` function, which passes flags that mention shared resources. For example, the `clone` function calls for a thread, which is `clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0)`, while for the normal fork looks similar to `clone(SIGCHLD, 0)`.

The notion of kernel threads appeared as a solution to problems involving tasks running in the background of the kernel context. The kernel thread does not have an address space and is only available inside the kernel context. It has the same properties as a normal process, but is only used for special tasks, such as `ksoftirqd`, `flush`, and so on.

At the end of the execution, the process need to be terminated so that the resources can be freed, and the parent of the executing process needs to be notified about this. The method that is most used to terminate a process is done by calling the `exit()` system call. A number of steps are needed for this process:

1. The `PF_EXITING` flag is set.
2. The `del_timer_sync()` function is called to remove the kernel timers.
3. The `acct_update_integrals()` function is called when writing accounting and logging information.
4. The `exit_mm()` is called to release the `mm_struct` structure for the process.
5. The `exit_sem()` is called to dequeue the process from the IPC semaphore.
6. The `exit_files()` and `exit_fs()` function are called to remove the links to various files descriptors.
7. The task exit code should be set.
8. Call `exit_notify()` to notify the parent and set the task exit state to `EXIT_ZOMBIE`.
9. Call `schedule()` to switch to a new process.

After the preceding steps are performed, the object associated with this task is freed and it becomes unrunnable. Its memory exists solely as information for its parent. After its parent announces that this information is of no use to it, this memory is freed for the system to use.

Process scheduling

The process scheduler decides which resources are allocated for a runnable process. It is a piece of software that is responsible for multitasking, resource allocation to various processes, and decides how to best set the resources and processor time. it also decides which processes should run next.

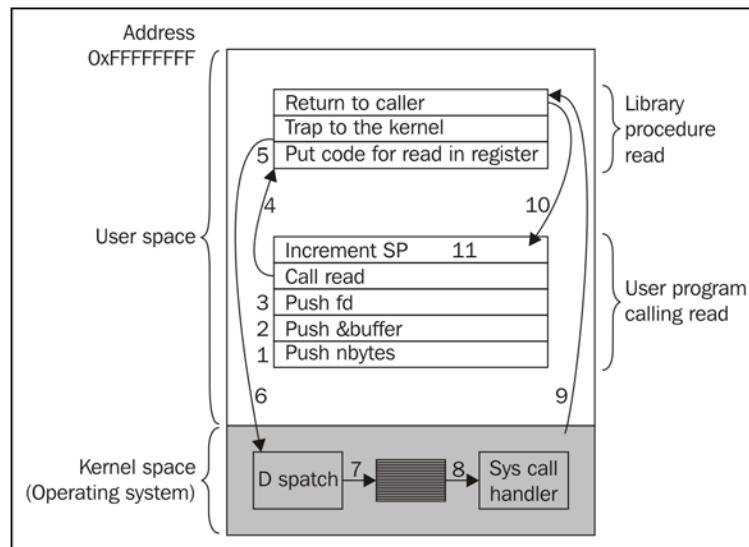
The first design of the Linux scheduler was very simplistic. It was not able to scale properly when the number of processes increased, so from the 2.5 kernel version, a new scheduler was developed. It is called **O(1) scheduler** and offers a constant time algorithm for time slice calculation and a run queue that is defined on a per-processor basis. Although it is perfect for large servers, it is not the best solution for a normal desktop system. From the 2.6 kernel version, improvements have been made to the O(1) scheduler, such as the fair scheduling concept that later materialized from the kernel version 2.6.23 into the **Completely Fair Scheduler (CFS)**, which became the defacto scheduler.

The CFC has a simple idea behind. It behaves as if we have a perfect multitasking processor where each process gets $1/n$ slice of the processor's time and this time slice is an incredibly small. The n value represents the number of running processes. Con Kolivas is the Australian programmer that contributed to the fair scheduling implementation, also known as **Rotating Staircase Deadline Scheduler (RSDL)**. Its implementation required a red-black tree for the priorities of self-balancing and also a time slice that is calculated at the nanosecond level. Similarly to the $O(1)$ scheduler, CFS applies the notion of weight, which implies that some processes wait more than others. This is based on the weighed fair queuing algorithm.

A process scheduler constitutes one of the most important components of the Linux kernel because it defines the user interaction with the operating system in general. The Linux kernel CFS is the scheduler that appeals to developers and users because it offers scalability and performance with the most reasonable approach.

System calls

For processes to interact with a system, an interface should be provided to give the user space application the possibility of interacting with hardware and other processes. System calls. These are used as an interface between the hardware and the user space. They are also used to ensure stability, security, and abstraction, in general. These are common layers that constitute an entry point into the kernel alongside traps and exceptions, as described here:



The interaction with most of the system calls that are available inside the Linux system is done using the C library. They are able to define a number of arguments and return a value that reveals whether they were successful or not. A value of `zero` usually means that the execution ended with success, and in case errors appear, an error code will be available inside the `errno` variable. When a system call is done, the following steps are followed:

1. The switch into kernel mode is made.
2. Any restrictions to the kernel space access are eliminated.
3. The stack from the user space is passed into the kernel space.
4. Any arguments from the user space are checked and copied into the kernel space.
5. The associated routine for the system call is identified and run.
6. The switch to the user space is made and the execution of the application continues.

A system call has a `syscall` number associated with it, which is a unique number used as a reference for the system call that cannot be changed (there is no possibility of implementing a system call). A symbolic constant for each system call number is available in the `<sys/syscall.h>` header file. To check the existence of a system call, `sys_ni_syscall()` is used, which returns the `ENOSYS` error for an invalid system call.

The virtual file system

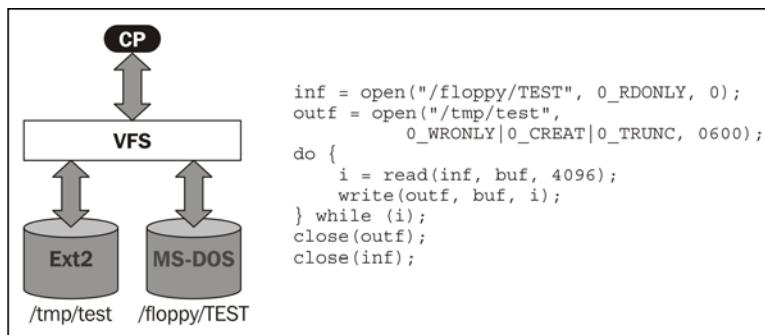
The Linux operating system is able to support a large variety of filesystem options. This is done due to the existence of **Virtual File System (VFS)**, which is able to provide a common interface for a large number of filesystem types and handle the systems calls relevant to them.

The filesystem types supported by the VFS can be put in these three categories:

- **Disk-based filesystems:** These manage the memory on a local disk or devices that are used for disk emulation. Some of the most well known ones are:
 - Linux filesystems, such as Second Extended Filesystem (Ext2), Third Extended Filesystem (Ext3), and Forth Extended Filesystem (Ext4)
 - UNIX filesystems, such as sysv filesystem, UFS, Minix filesystem, and so on
 - Microsoft filesystems, such as MS-DOS, NTFS (available since Windows NT), and VFAT (available since Windows 95)

- ISO966 CD-ROM filesystem and disk format DVD filesystem
- Proprietary filesystems, such as the ones from Apple, IBM, and other companies
- **Network filesystems:** They are allowed to access various filesystem types over a network on other computers. One of the most well known ones is NFS. Of course, there are others but they are not as well known. These include **Andrew filesystem (AFS)**, **Novel's NetWare Core Protocol (NCP)**, **Constant Data Availability (Coda)**, and so on.
- **Special filesystems:** The /proc filesystem is the perfect example for this category of filesystems. This category of filesystems enables an easier access for system applications to interrogate data structures of kernels and implement various features.

The virtual filesystem system call implementation is very well summarized in this image:



In the preceding image, it can be seen how easily the copy is handled from one filesystem type to another. It only uses the basic `open()`, `close()`, `read()`, `write()` functions available for all the other filesystem interaction. However, all of them implement the specific functionality underneath for the chosen filesystem. For example, the `open()` system calls `sys_open()` and it takes the same arguments as `open()` and returns the same result. The difference between `sys_open()` and `open()` is that `sys_open()` is a more permissive function.

All the other three system calls have corresponding `sys_read()`, `sys_write()`, and `sys_close()` functions that are called internally.

Interrupts

An interrupt is a representation of an event that changes the succession of instructions performed by the processor. Interrupts imply an electric signal generated by the hardware to signal an event that has happened, such as a key press, reset, and so on. Interrupts are divided into more categories depending on their reference system, as follows:

- Software interrupts: These are usually exceptions triggered from external devices and user space programs
- Hardware interrupts: These are signals from the system that usually indicate a processor specific instruction

The Linux interrupt handling layer offers an abstraction of interrupt handling for various device drivers through comprehensive API functions. It is used to request, enable, disable, and free interrupts, making sure that portability is guaranteed on multiple platforms. It handles all available interrupt controller hardware.

The generic interrupt handling uses the `__do_IRQ()` handler, which is able to deal with all the available types of the interrupt logic. The handling layers are divided in two components:

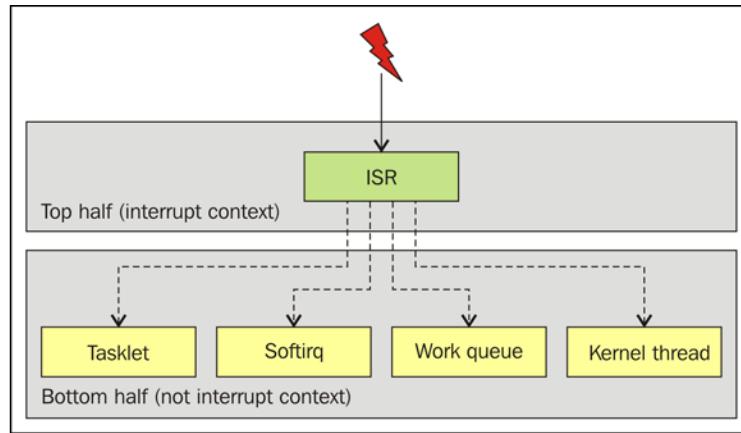
- The top half component is used to respond to the interrupt
- The bottom half component is scheduled by the top half to run at a later time

The difference between them is that all the available interrupts are permitted to act in the bottom half context. This helps the top half respond to another interrupt while the bottom half is working, which means that it is able to save its data in a specific buffer and it permits the bottom half to operate in a safe environment.

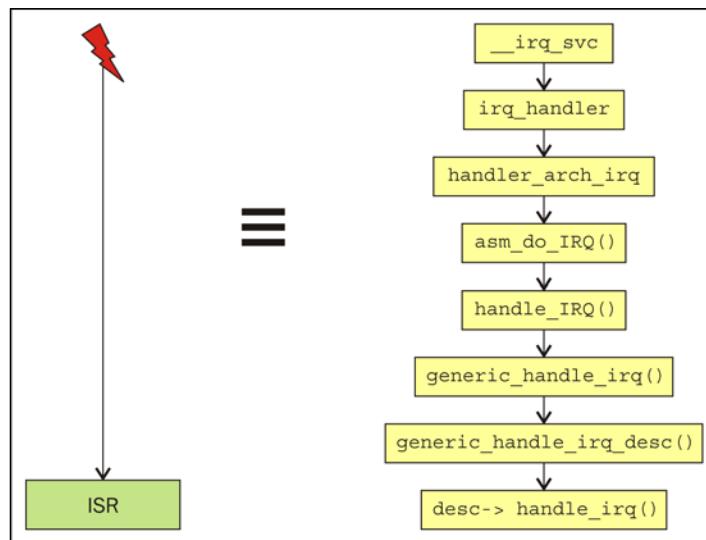
For the bottom half processing, there are four defined mechanisms available:

- **Softirqs**
- **Tasklets**
- **Work queues**
- **Kernel threads**

The available mechanisms are well presented here:



Although the model for the top and bottom half interrupt mechanism looks simple, it has a very complicated function calling mechanism model. This example shows this fact for the ARM architecture:



For the top half component of the interrupt, there are three levels of abstraction in the interrupt source code. The first one is the high-level driver API that has functions, such as `request_irq()`, `free_irq`, `disable_irq()`, `enable_irq()`, and so on. The second one is represented by the high-level IRQ flow handlers, which is a generic layer with predefined or architecture-specific interrupt flow handlers assigned to respond to various interrupts during device initialization or boot time. It defines a number of predefined functions, such as `handle_level_irq()`, `handle_simple_irq()`, `handle_percpu_irq()`, and so on. The third is represented by chip-level hardware encapsulation. It defines the `struct irq_chip` structure that holds chip-relevant functions used in the IRQ flow implementation. Some of the functions are `irq_ack()`, `irq_mask()`, and `irq_unmask()`.

A module is required to register an interrupt channel and release it afterwards. The total number of supported requests is counted from the 0 value to the number of IRQs-1. This information is available inside the `<asm/irq.h>` header file. When the registration is done, a handler flag is passed to the `request_irq()` function to specify the interrupt handler's type, as follows:

- `SA_SAMPLE_RANDOM`: This indicates that the interrupt can contribute to the entropy pool, that is, a pool with bits that possess a strong random property, by sampling unpredictable events, such as mouse movement, inter-key press time, disk interrupts, and so on
- `SA_SHIRQ`: This shows that the interrupt is sharable between devices.
- `SA_INTERRUPT`: This indicates a fast interrupt handler, so interrupts are disabled on the current processor-it does not represent a situation that is very desirable

Bottom halves

The first mechanism that will be discussed regarding bottom half interrupt handling is represented by `softirqs`. They are rarely used but can be found on the Linux kernel source code inside the `kernel/softirq.c` file. When it comes to implementation, they are statically allocated at the compile step. They are created when an entry is added in the `include/linux/interrupt.h` header file and the system information they provide is available inside the `/proc/softirqs` file. Although not used too often, they can be executed after exceptions, interrupts, system calls, and when the `ksoftirqd` daemon is run by the scheduler.

Next on the list are tasklets. Although they are built on top of softirqs, they are more commonly used for bottom half interrupt handling. Here are some of the reasons why this is done:

- They are very fast
- They can be created and destroyed dynamically
- They have atomic and nonblocking code
- They run in a soft interrupt context
- They run on the same processor that they were scheduled for

Tasklets have a **struct tasklet_struct** structure available. These are also available inside the `include/linux/interrupt.h` header file, and unlike softirqs, tasklets are non-reentrant.

Third on the list are work queues that represent a different form of doing the work allotted in comparison to previously presented mechanisms. The main differences are as follows:

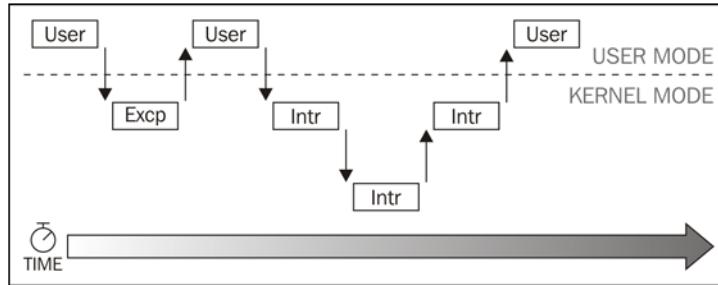
- They are able run in the same time on more than one CPU
- They are allowed to go to sleep
- They run on a process context
- They can be scheduled or preempted

Although they might have a latency that is slightly bigger than the tasklets, the preceding qualities are really useful. The tasklets are built around the `struct workqueue_struct` structure, available inside the `kernel/workqueue.c` file.

The last and the newest addition to the bottom half mechanism options is represented by the kernel threads that are operated entirely in the kernel mode since they are created/destroyed by the kernel. They appeared during the 2.6.30 kernel release, and also have the same advantages as the work queues, along with some extra features, such as the possibility of having their own context. It is expected that eventually the kernel threads will replace the work queues and tasklets, since they are similar to the user space threads. A driver might want to request a threaded interrupt handler. All it needs to do in this case is to use `request_threaded_irq()` in a similar way to `request_irq()`. The `request_threaded_irq()` function offers the possibility of passing a handler and `thread_fn` to split the interrupt handling code into two parts. In addition to this, `quick_check_handler` is called to check if the interrupt was called from a device; if that is the case, it will need to call `IRQ_WAKE_THREAD` to wake up the handler thread and execute `thread_fn`.

Methods to perform kernel synchronization

The number of requests with which a kernel is dealing is likened to the number of requests a server has to receive. This situation can deal with race conditions, so a good synchronization method would be required. A number of policies are available for the way the kernel behaves by defining a kernel control path. Here is an example of a kernel control path:



The preceding image offers a clear picture as to why synchronization is necessary. For example, a race condition can appear when more than one kernel control path is interlinked. To protect these critical regions, a number of measures should be taken. Also, it should be taken into consideration that an interrupt handler cannot be interrupted and softirqs should not be interleaved.

A number of synchronization primitives have been born:

- **Per-CPU variables:** This is one of the most simple and efficient synchronization methods. It multiplies a data structure so that each one is available for each CPU.
- **Atomic operations:** This refers to atomic read-modify-write instructions.
- **Memory barrier:** This safeguards the fact that the operations done before the barrier are all finished before starting the operations after it.
- **Spin lock:** This represents a type of lock that implements bust waiting.
- **Semaphore:** This is a form of locking that implements sleep or blocking waiting.
- **Seqlocks:** This is similar to spin locks, but is based on an access counter.
- **Local interrupt disabling:** This forbids the use of functions that can be postponed on a single CPU.
- **Read-copy-update(RCU):** This is a method designed to protect the most used data structures used for reading. It offers a lock-free access to shared data structures using pointers.

With the preceding methods, race condition situations try to be fixed. It is the job of the developer to identify and solve all the eventual synchronization problems that might appear.

Timers

Around the Linux kernel, there are a great number of functions that are influenced by time. From the scheduler to the system uptime, they all require a time reference, which includes both absolute and relative time. For example, an event that needs to be scheduled for the future, represents a relative time, which, in fact, implies that there is a method used to count time.

The timer implementation can vary depending on the type of the event. The periodical implementations are defined by the system timer, which issues an interrupt at a fixed period of time. The system timer is a hardware component that issues a timer interrupt at a given frequency to update the system time and execute the necessary tasks. Another one that can be used is the real-time clock, which is a chip with a battery attached that keeps counting time long after the system was shut down. Besides the system time, there are dynamic timers available that are managed by the kernel dynamically to plan events that run after a particular time has passed.

The timer interrupt has an occurrence window and for ARM, it is 100 times per second. This is called the **system timer frequency** or **tick rate** and its unit of measurement is **hertz (Hz)**. The tick rate differs from one architecture to another. If for the most of them, we have the value of 100 Hz, there are others that have values of 1024 Hz, such as the Alpha and Itanium (IA-64) architectures, for example. The default value, of course, can be changed and increased, but this action has its advantages and disadvantages.

Some of the advantages of higher frequency are:

- The timer will be executed more accurately and in a larger number
- System calls that use a timeout are executed in a more precise manner
- Uptime measurements and other similar measurements are becoming more precise
- The preemption of process is more accurate

The disadvantages of higher frequency on the other hand, implies a higher overhead. The processors spend more time in a timer interrupt context; also, an increase in power consumption will take place because more computing is done.

The total number of ticks done on a Linux operation system from the time it started booting is stored in a variable called **jiffies** inside the `include/linux/jiffies.h` header file. At boot time, this variable is initialized to zero and one is added to its value each time an interrupt happens. So, the actual value of the system uptime can be calculated in the form of jiffies/Hz.

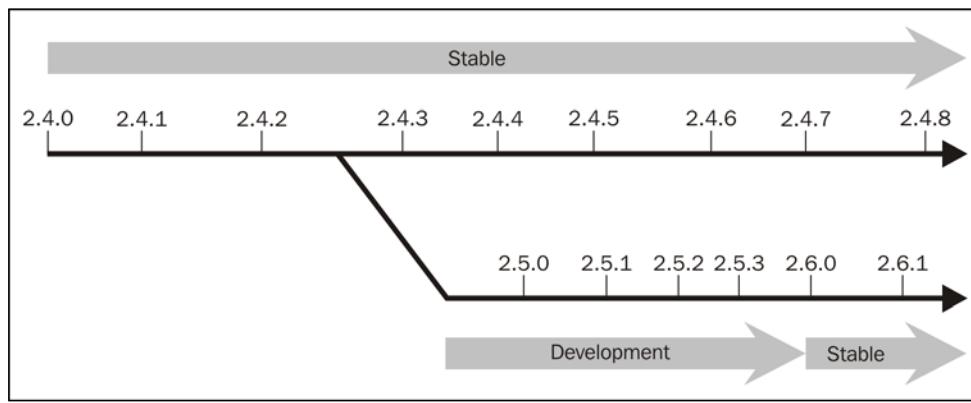
Linux kernel interaction

Until now, you were introduced to some of features of the Linux kernel. Now, it is time to present more information about the development process, versioning scheme, community contributions, and interaction with the Linux kernel.

The development process

Linux kernel is a well known open source project. To make sure that developers know how to interact with it, information about how the `git` interaction is done with this project, and at the same time, some information about its development and release procedures will be presented. The project has evolved and its development processes and release procedures have evolved with it.

Before presenting the actual development process, a bit of history will be necessary. Until the 2.6 version of the Linux kernel project, one release was made every two or three years, and each of them was identified by even middle numbers, such as 1.0.x, 2.0.x, and 2.6.x. The development branches were instead defined using even numbers, such as 1.1.x, 2.1.x, and 2.5.x, and they were used to integrate various features and functionalities until a major release was prepared and ready to be shipped. All the minor releases had names, such as 2.6.32 and 2.2.23, and they were released between major release cycles.



This way of working was kept up until the 2.6.0 version when a large number of features were added inside the kernel during every minor release, and all of them were very well put together as to not cause the need for the branching out of a new development branch. This implied a faster pace of release with more features available. So, the following changes have appeared since the release of the 2.6.14 kernel:

- All the new minor release versions, such as 2.6.x, contain a two week merge window in which a number of features could be introduced in the next release
- This merge window will be closed with a release test version called 2.6.(x+1)-rc1
- Then a 6-8 weeks bug fixing period follows when all the bugs introduced by the added features should be fixed
- In the bug fixing interval, tests were run on the release candidate and the 2.6.(x+1)-rcY test versions were released
- After the final test were done and the last release candidate is considered sufficiently stable, a new release will be made with a name, such as 2.6.(x+1), and this process will be continued once again

This process worked great but the only problem was that the bug fixes were only released for the latest stable versions of the Linux kernel. People needed long term support versions and security updates for their older versions, general information about these versions that were long time supported, and so on.

This process changed in time and in July 2011, the 3.0 Linux kernel version appeared. It appeared with a couple of small changes designed to change the way the interaction was to be done to solve the previously mentioned requests. The changes were made to the numbering scheme, as follows:

- The kernel official versions would be named 3.x (3.0, 3.1, 3.2, and so on)
- The stable versions would be named 3.x.y (3.0.1, 3.1.3, and so on)

Although it only removed one digit from the numbering scheme, this change was necessary because it marked the 20th anniversary of the Linux kernel.

Since a great number of patches and features are included in the Linux kernel everyday, it becomes difficult to keep track of all the changes, and the bigger picture in general. This changed over time because sites, such as <http://kernelnewbies.org/LinuxChanges> and <http://lwn.net/>, appeared to help developers keep in touch with the world of Linux kernel.

Besides these links, the `git` versioning control system can offer much needed information. Of course, this requires the existence of Linux kernel source clones to be available on the workstation. Some of the commands that offer a great deal of information are:

- `git log`: This lists all the commits with the latest situated on top of the list
- `git log -p`: This lists all the commits and with their corresponding `diffs`
- `git tag -l`: This lists the available tags
- `git checkout <tagname>`: This checks out a branch or tag from a working repository
- `git log v2.6.32..master`: This lists all the changes between the given tag and the latest version
- `git log -p V2.6.32..master MAINTAINERS`: This lists all the differences between the two given branches in the `MAINTAINERS` file

Of course, this is just a small list with helpful commands. All the other commands are available at <http://git-scm.com/docs/>.

Kernel porting

The Linux kernel offers support for a large variety of CPU architectures. Each architecture and individual board have their own maintainers, and this information is available inside the `MAINTAINERS` file. Also, the difference between board porting is mostly given by the architecture, PowerPC being very different from ARM or x86. Since the development board that this book focuses on is an Atmel with an ARM Cortex-A5 core, this section will try to focus on ARM architecture.

The main focus in our case is the `arch/arm` directory, which contains sub directories such as, `boot`, `common`, `configs`, `crypto`, `firmware`, `kernel`, `kvm`, `lib`, `mm`, `net`, `nwfpe`, `oprofile`, `tools`, `vfp`, and `xen`. It also contains an important number of directories that are specific for different CPU families, such as the `mach-*` directories or the `plat-*` directories. The first `mach-*` category contains support for the CPU and several boards that use that CPU, and the second `plat-*` category contains platform-specific code. One example is `plat-omap`, which contains common code for both `mach-omap1` and `mach-omap2`.

The development for the ARM architecture has suffered a great change since 2011. If until then ARM did not use a device tree, it was because it needed to keep a large portion of the code inside the `mach-*` specific directory, and for each board that had support inside the Linux kernel, a unique machine ID was associated and a machine structure was associated with each board that contained specific information and a set of callbacks. The boot loader passed this machine ID to a specific ARM registry and in this way, the kernel knew the board.

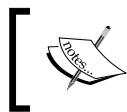
The increase in popularity of the ARM architecture came with the refactoring of the work and the introduction of the **device tree** that dramatically reduced the amount of code available inside the `mach-*` directories. If the SoC is supported by the Linux kernel, then adding support for a board is as simple as defining a device tree in the `/arch/arm/boot/dts` directory with an appropriate name. For example, for `<soc-name>-<board-name>.d`, include the relevant `dtsi` files if necessary. Make sure that you build the **device tree blob (DTB)** by including the device tree into `arch/arm/boot/dts/Makefile` and add the missing device drivers for board.

In the eventuality that the board does not have support inside the Linux kernel, the appropriate additions would be required inside the `mach-*` directory. Inside each `mach-*` directory, there are three types of files available:

- **Generic code files:** These usually have a single word name, such as `clock.c`, `led.c`, and so on
- **CPU specific code:** This is for the machine ID and usually has the `<machine-ID>*.c` form - for example, `at91sam9263.c`, `at91sam9263_devices.c`, `sama5d3.c`, and so on
- **Board specific code:** This usually is defined as `board-*.c`, such as `board-carmeva.c`, `board-pcontrol-g20.c`, `board-pcontrol-g20.c`, and so on

For a given board, the proper configuration should be made first inside the `arch/arm/mach-*/Kconfig` file; for this, the machine ID should be identified for the board CPU. After the configuration is done, the compilation can begin, so for this, `arch/arm/mach-*/Makefile` should also be updated with the required files to ensure board support. Another step is represented by the machine structure that defines the board and the machine type number that needs to be defined in the `board-<machine>.c` file.

The machine structure uses two macros: `MACHINE_START` and `MACHINE_END`. Both are defined inside `arch/arm/include/asm/mach/arch.h` and are used to define the `machine_desc` structure. The machine type number is available inside the `arch/arm/tools/mach_types` file. This file is used to generate the `include/asm-arm/mach-types.h` file for the board.



The updated number list of the machine type is available at <http://www.arm.linux.org.uk/developer/machines/download.php>.

When the boot process starts in the first case, only the `dtb` is necessary to pass to the boot loader and loaded to initialize the Linux kernel, while in the second case, the machine type number needs to be loaded in the `R1` register. In the early boot process, `_lookup_machine_type` looks for the `machine_desc` structure and loads it for the initialization of the board.

Community interaction

After this information has been presented to you, and if you are eager to contribute to the Linux kernel, then this section should be read next. If you want to really contribute to the Linux kernel project, then a few steps should be performed before starting this work. This is mostly related to documentation and investigation of the subject. No one wants to send a duplicate patch or replicate the work of someone else in vain, so a search on the Internet on the topic of your interest could save a lot of time. Other useful advice is that after you've familiarized yourself with the subject, avoid sending a workaround. Try to reach the problem and offer a solution. If not, report the problem and describe it thoroughly. If the solution is found, then make both the problem and solution available in the patch.

One of the most valuable things in the open source community is the help you can get from others. Share your question and issues, but do not forget to mention the solution also. Ask the questions in appropriate mailing lists and try to avoid the maintainers, if possible. They are usually very busy and have hundreds and thousands of e-mails to read and reply. Before asking for help, try to research the question you want to raise, it will help both when formulating it but also it could offer an answer. Use IRC, if available, for smaller questions and lastly, but most importantly, try to not overdo it.

When you are preparing for a patch, make sure that it is done on the corresponding branch, and also that you read the `Documentation/BUG-HUNTING` file first. Identify bug reports, if any, and make sure you link your patch to them. Do not hesitate to read the `Documentation/SubmittingPatches` guidelines before sending. Also, do not send your changes before testing them properly. Always sign your patches and make the first description line as suggestive as possible. When sending the patches, find appropriate mailing lists and maintainers and wait for the replies. Solve comments and resubmit them if this is needed, until the patch is considered acceptable.

Kernel sources

The official location for the Linux kernel is available at <http://www.kernel.org>, but there are a lot of smaller communities that contribute to the Linux kernel with their features or even maintain their own versions.

Although the Linux core contains the scheduler, memory management, and other features, it is quite small in size. The extremely large number of device drivers, architectures and boards support together with filesystems, network protocols and all the other components were the ones that made the size of the Linux kernel really big. This can be seen by taking a look at the size of the directories of the Linux.

The Linux source code structure contains the following directories:

- `arch`: This contains architecture-dependent code
- `block`: This contains the block layer core
- `crypto`: This contains cryptographic libraries
- `drivers`: This gathers all the implementation of the device drivers with the exception of the sound ones
- `fs`: This gathers all the available implementations of filesystem
- `include`: This contains the kernel headers
- `init`: This has the Linux initialization code
- `ipc`: This holds the interprocess communication implementation code
- `kernel`: This is the core of the Linux kernel
- `lib`: This contains various libraries, such as `zlibc`, `crc`, and so on
- `mm`: This contains the source code for memory management
- `net`: This offers access to all the network protocol implementations supported inside Linux
- `samples`: This presents a number of sample implementations, such as `kfifo`, `kobject`, and so on
- `scripts`: This is used both internally and externally
- `security`: This has a bunch of security implementation, such as `apparmor`, `selinux`, `smack`, and so on
- `sound`: This contains sound drivers and support code
- `usr`: This is the `initramfs cpio` archive that generates sources
- `virt`: This holds the source code for the virtualization support
- `COPYING`: This represents the Linux license and the definition copying conditions

- **CREDITS:** This represents the collection of Linux's main contributors
- **Documentation:** This contains corresponding documentation of kernel sources
- **Kbuild:** This represents the top-level kernel build system
- **Kconfig:** This is the top-level descriptor for configuration parameters
- **MAINTAINERS:** This a list with the maintainers of each kernel component
- **Makefile:** This represents the top-level makefile
- **README:** This file describes what Linux is, it is the starting point for understanding the project
- **REPORTING-BUGS:** This offers information regarding the bug report procedure

As it can be seen, the source code of the Linux kernel is quite large, so a browsing tool would be required. There are a number of tools that can be used, such as **Cscope**, **Kscope**, or the web browser, **Linux Cross Reference (LXR)**. Cscope is a huge project that can be also available with extensions for `vim` and `emacs`.

Configuring kernel

Before building a Linux kernel image, the proper configuration needs to be done. This is hard, taking into consideration that we have access to hundreds and thousands of components, such as drivers, filesystems, and other items. A selection process is done inside the configuration stage, and this is possible with the help of dependency definitions. The user has the chance to use and define a number of options that are enabled in order to define the components that will be used to build a Linux kernel image for a specific board.

All the configurations specific for a supported board are located inside a configuration file, simply named `.config`, and it is situated on the same level as the previously presented files and directory locations. Their form is usually represented as `configuration_key=value`. There are, of course, dependencies between these configurations, so they are defined inside the `Kconfig` files.

Here are a number of variable options available for a configuration key:

- **bool:** These are the options can have true or false values
- **tristate:** This, besides the true and false options, also appears as a module option
- **int :** These values, are not that spread but they usually have a well-established value range
- **string :** These values, are also not the most spread ones but usually contain some pretty basic information

With regard to the `Kconfig` files, there are two options available. The first one makes option A visible only when option B is enabled and is defined as *depends on*, and the second option offers the possibility of enabling option A. This is done when the option is enabled automatically and is defined as *select*.

Besides the manual configuration of the `.config` file, configuration is the worst option for a developer, mostly because it can miss dependencies between some of the configurations. I would like to suggest to developers to use the `make menuconfig` command that will launch a text console tool for the configuration of a kernel image.

Compiling and installing the kernel

After the configuration is done, the compilation process can be started. A piece of advice I would like to give is to use as many threads as possible if the host machine offers this possibility because it would help with the build process. An example of the build process start command is `make -j 8`.

At the end of the build process, a `vmlinux` image is offered and also some architecture-dependent images are made available inside the architecture-specific files for the ARM architecture. The result of this is available inside `arch/arm/boot/*Image`. Also, the Atmel SAMA5D3-Xplained board will offer a specific device tree file that is available in `arch/arm/boot/dts/* .dtb`. If the `vmlinux` image file is an ELF file with debug information that cannot be used for booting except for debug purposes, the `arch/arm/boot/*Image` file is the solution for this purpose.

The installation is the next step when development is done for any other application. The same also takes place for the Linux kernel, but in an embedded environment, this step seems kind of unnecessary. For Yocto enthusiasts, this step is also available. However, in this step, proper configurations are done for the kernel source and headers are to be used by the dependencies that do the storing for the deploy step.

The kernel modules, as mentioned in the cross-compilation chapter, need to be later used for the compiler build. The install for the kernel modules could be done using the `make modules_install` command, and this offers the possibility to install the sources available inside the `/lib/modules/<linux-kernel-version>` directory with all the module dependencies, symbols, and aliases.

Cross-compiling the Linux kernel

In an embedded development, the compilation process implies cross-compilation, the most visible difference with the native compilation process being the fact that it has a prefix with the target architecture available in the naming. The prefix setup can be done using the `ARCH` variable that defines the name of the architecture of the target board and the `CROSS_COMPILE` variable that defines the prefix for the cross-compilation toolchain. Both of them are defined in the top-level `Makefile`.

The best option would be to set these variables as environment variables to make sure that a make process is not run for the host machine. Although it only works in the current terminal, it will be the best solution in the situation that no automation tool is available for these tasks, such as the Yocto Project. It is not recommended though to update the `.bashrc` shell variables if you are planning to use more than one toolchain on the host machine.

Devices and modules

As I mentioned previously, the Linux kernel has a lot of kernel modules and drivers that are already implemented and available inside the source code of the Linux kernel. A number of them, being so many, are also available outside the Linux kernel source code. Having them outside not only reduces the boot time by not initializing them at boot time, but is done instead at the request and needs of users. The only difference is that the loading and unloading of the modules requires root access.

Loading and interacting with the Linux kernel modules requires logging information to be made available. The same happens for any kernel module dependencies. The logging information is available through the `dmesg` command and the level of logging enables manual configuration using the `loglevel` parameter or it can be disabled with the `quiet` parameter. Also for the kernel dependencies, information about them is available inside the `/lib/modules/<kernel-version>/modules.dep` file.

For module interaction, multiple utilities used for multiple operations are available, such as `modinfo`, which is used for information gathering about modules; `insmod` is able for loading a module when the full path to the kernel module is given. Similar utilities for a module are available. One of them is called `modprobe` and the difference in `modprobe` is that the full path is not necessary, as it is responsible for loading dependent modules of the chosen kernel object before loading itself. Another functionality that `modprobe` offers is the `-r` option. It is the remove functionality which offers support for removing the module and all its dependencies. An alternative to this is the `rmmmod` utility, which removes modules not used anymore. The last utility available is `lsmod`, which lists the loaded modules.

The simplest kernel module example that can be written looks something similar to this:

```
#define MODULE
#define LINUX
#define __KERNEL__

#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

static int hello_world_init(void)
{
    printk(KERN_ALERT "Hello world!\n");
    return 0;
}

static void hello_world_exit(void)
{
    printk(KERN_ALERT "Goodbye!\n");
}

module_init(hello_world_init);
module_exit(hello_world_exit);

MODULE_LICENSE("GPL");
```

This is a simple `hello world` kernel module. Useful information that can be gathered from the preceding example is that every kernel module needs a start function defined in the preceding example as `hello_world_init()`. It is called when the module is inserted, and a cleanup function called `hello_world_exit()` is called when the module is removed.

Since the Linux kernel version 2.2, there is a possibility of using the `_init` and `_exit` macros in this way:

```
static int __init hello_world_init (void)
static void __exit hello_world_exit (void)
```

The preceding macros are removed, the first one after the initialization, and the second one when the module is built-in within the Linux kernel sources.



More information about the Linux kernel modules can be found in the **Linux Kernel Module Programming Guide** available at <http://www.tldp.org/LDP/lkmpg/2.6/html/index.html>.

As mentioned previously, a kernel module is not only available inside a Linux kernel, but also outside of the Linux kernel tree. For a built-in kernel module, the compile process is similar to the one of other available kernel modules and a developer can inspire its work from one of them. The kernel module available outside of the Linux kernel drivers and the build process requires access to the sources of the Linux kernel or the kernel headers.

For a kernel module available outside of the Linux kernel sources, a `Makefile` example is available, as follows:

```
KDIR := <path/to/linux/kernel/sources>
PWD := $(shell pwd)
obj-m := hello_world.o
all:
$(MAKE) ARCH=arm CROSS_COMPILE=<arm-cross-compiler-prefix> -C
$(KDIR) M=$(PWD)
```

For a module that is implemented inside a Linux kernel, a configuration for the module needs to be made available inside the corresponding `Kconfig` file with the correct configuration. Also, the `Makefile` near the `Kconfig` file needs to be updated to let the `Makefile` system know when the configuration for the module is updated and the sources need to be built. We will see an example of this kind for a kernel device driver here.

An example of the `Kconfig` file is as follows:

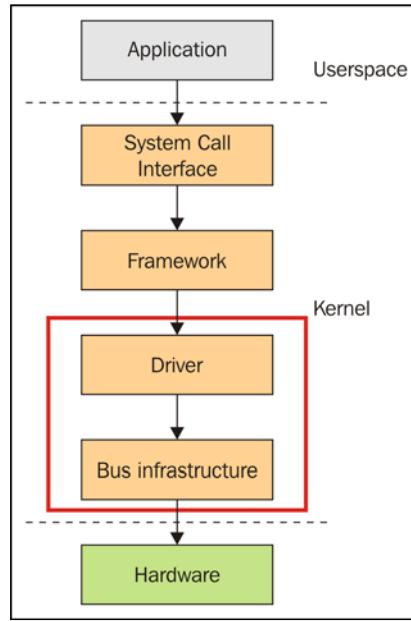
```
config HELLO_WORLD_TEST
tristate "Hello world module test"
help
To compile this driver as a module chose the M option.
otherwise chose Y option.
```

An example of the `Makefile` is as follows:

```
obj-$(CONFIG_HELLO_WORLD_TEST) += hello_world.c
```

In both these examples, the source code file is `hello_world.c` and the resulting kernel module if it is not built-in is called `hello_world.ko`.

A driver is usually used as an interface with a framework that exposes a number of hardware features, or with a bus interface used to detect and communicate with the hardware. The best example is shown here:



Since there are multiple scenarios of using a device driver and three device mode structures are available:

- `struct bus_type`: This represents the types of busses, such as I2C, SPI, USB, PCI, MMC, and so on
- `struct device_driver`: This represents the driver used to handle a specific device on a bus
- `struct device`: This is used to represent a device connected to a bus

An inheritance mechanism is used to create specialized structures from more generic ones, such as `struct device_driver` and `struct device` for every bus subsystem. The bus driver is the one responsible for representing each type of bus and matching the corresponding device driver with the detected devices, detection being done through an adapter driver. For nondiscoverable devices, a description is made inside the device tree or the source code of the Linux kernel. They are handled by the platform bus that supports platform drivers and in return, handles platform devices.

Debugging a kernel

Having to debug the Linux kernel is not the most easy task, but it needs to be accomplished to make sure that the development process moves forward.

Understanding the Linux kernel is, of course, one of the prerequisites. Some of the available bugs are very hard to solve and may be available inside the Linux kernel for a long period of time.

For most of the trivial ones, some of the following steps should be taken. First, identify the bug properly; it is not only useful when define the problem, but also helps with reproducing it. The second step involves finding the source of the problem. Here, I am referring to the first kernel version in which the bug was first reported. Good knowledge about the bug or the source code of the Linux kernel is always useful, so make sure that you understand the code before you start working on it.

The bugs inside the Linux kernel have a wide spread. They vary from a variable not being stored properly to race conditions or hardware management problems, they have widely variable manifestations and a chain of events. However, debugging them is not as difficult as it sounds. Besides some specific problems, such as race conditions and time constraints, debugging is very similar to the debugging of any large user space application.

The first, easiest, and most handy method to debug the kernel is the one that involves the use of the `printk()` function. It is very similar to the `printf()` C library function, and although old and not recommended by some, it does the trick. The new preferred method involves the usage of the `pr_*`() functions, such as `pr_emerg()`, `pr_alert()`, `pr_crit()`, `pr_debug()`, and so on. Another method involves the usage of the `dev_*`() functions, such as `dev_emerg()`, `dev_alert()`, `dev_crit()`, `dev_dbg()`, and so on. They correspond to each logging level and also have extra functions that are defined for debugging purposes and are compiled when `CONFIG_DEBUG` is enabled.



More information about the `pr_*`() and `dev_*`() family of functions can be found inside the Linux kernel source code at `Documentation/dynamic-debug-howto.txt`. You can also find more information about `loglevel` at `Documentation/kernel-parameters.txt`.

When a kernel **oops** crash appears, it signals that the kernel has made a mistake. Not being able to fix or kill itself, it offers access to a bunch of information, such as useful error messages, registers content, and back trace information.

The `Magic SysRq` key is another method used in debugging. It is enabled by `CONFIG_MAGIC_SYSRQ` config and can be used to debug and rescue kernel information, regardless of its activity. It offers a series of command-line options that can be used for various actions, ranging from changing the nice level to rebooting the system. Plus, it can be toggled on or off by changing the value in the `/proc/sys/kernel/sysrq` file. More information about the system request key can be found at [Documentation/sysrq.txt](#).

Although Linus Torvalds and the Linux community do not believe that the existence of a kernel debugger will do much good to a project, a better understanding of the code is the best approach for any project. There are still some debugger solutions that are available to be used. GNU debugger (`gdb`) is the first one and it can be used in the same way as for any other process. Another one is the `kgdb` a patch over `gdb` that permits debugging of serial connections.

If none of the preceding methods fail to help solve the problem and you've tried everything but can't seem to arrive at a solution, then you can contact the open source community for help. There will always will be developers there who will lend you a hand.



To acquire more information related to the Linux kernel, there are a couple of books that can be consulted. I will present a bunch of their names here: *Embedded Linux Primer* by Christopher Hallinan, *Linux Kernel Development* by Robert Love, *Linux Kernel In A Nutshell* by Greg Kroah-Hartman, and last but not the least, *Understanding the Linux Kernel* by Daniel P. Bovet and Marco Cesati.

The Yocto Project reference

Moving on to the Yocto Project, we have recipes available for every kernel version available inside the BSP support for each supported board, and recipes for kernel modules that are built outside the Linux kernel source tree.

The Atmel SAMA5D3-Xplained board uses the `linux-yocto-custom` kernel. This is defined inside the `conf/machine/sama5d3-xplained.conf` machine configuration file using the `PREFERRED_PROVIDER_virtual/kernel` variable. No `PREFERRED_VERSION` is mentioned, so the latest version is preferred; in this case, we are talking about the `linux-yocto-custom_3.10.bb` recipe.

The `linux-yocto-custom_3.10.bb` recipe fetches the kernel sources available inside Linux Torvalds' git repository. After a quick look at the sources once the `do_fetch` task is finished, it can be observed that the Atmel repository was, in fact, fetched. The answer is available inside the `linux-yocto-custom_3.10.bbappend` file, which offers another `SRC_URI` location. Other useful information you can gather from here is the one available in `bbappend` file, inside it is very well stated that the SAMA5D3 Xplained machine is a `COMPATIBLE_MACHINE`:

```
KBRANCH = "linux-3.10-at91"
SRCREV = "35158dd80a94df2b71484b9ffa6e642378209156"
PV = "${LINUX_VERSION}+${SRCPV}"

PR = "r5"

FILESEXTRAPATHS_prepend := "${THISDIR}/files/${MACHINE}:"

SRC_URI = "git://github.com/linux4sam/linux-at91.git;protocol=git;branch=${KBRANCH};noccheckout=1"
SRC_URI += "file://defconfig"

SRCREV_sama5d4-xplained = "46f4253693b0ee8d25214e7ca0dde52e788ffe95"

do_deploy_append() {
    if [ ${UBOOT_FIT_IMAGE} = "xyes" ] ; then
        DTB_PATH="${B}/arch/${ARCH}/boot/dts/"
        if [ ! -e "${DTB_PATH}" ] ; then
            DTB_PATH="${B}/arch/${ARCH}/boot/"
        fi

        cp ${S}/arch/${ARCH}/boot/dts/${MACHINE}*.its ${DTB_PATH}
        cd ${DTB_PATH}
        mkimage -f ${MACHINE}.its ${MACHINE}.itb
        install -m 0644 ${MACHINE}.itb ${DEPLOYDIR}/${MACHINE}.itb
        cd -
    fi
}

COMPATIBLE_MACHINE = "(sama5d4ek|sama5d4-xplained
|sama5d3xek|sama5d3-xplained|at91sam9x5ek
|at91sam9rlek|at91sam9m10g45ek)"

```

The recipe firstly defines repository-related information. It is defined through variables, such as `SRC_URI` and `SRCREV`. It also indicates the branch of the repository through the `KBRANCH` variable, and also the place from where `defconfig` needs to be put into the source code to define the `.config` file. As seen in the recipe, there is an update made to the `do_deploy` task for the kernel recipe to add the device driver to the `tmp/deploy/image/sama5d3-xplained` directory alongside the kernel image and other binaries.

The kernel recipe inherits the `kernel.bbclass` and `kernel-yocto.bbclass` files, which define most of its tasks actions. Since it also generates a device tree, it needs access to `linux-dtb.inc`, which is available inside the `meta/recipes-kernel/linux` directory. The information available in the `linux-yocto-custom_3.10.bb` recipe is rather generic and overwritten by the `bbappend` file, as can be seen here:

```
SRC_URI = "git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git; protocol=git; noccheckout=1"

LINUX_VERSION ?= "3.10"
LINUX_VERSION_EXTENSION ?= "-custom"

inherit kernel
require recipes-kernel/linux/linux-yocto.inc

# Override SRCREV to point to a different commit in a bbappend
# file to
# build a different release of the Linux kernel.
# tag: v3.10 8bb495e3f02401ee6f76d1b1d77f3ac9f079e376"
SRCREV = "8bb495e3f02401ee6f76d1b1d77f3ac9f079e376"

PR = "r1"
PV = "${LINUX_VERSION}+git${SRCPV}"

# Override COMPATIBLE_MACHINE to include your machine in a bbappend
# file. Leaving it empty here ensures an early explicit build
failure.
COMPATIBLE_MACHINE = "(^$)"

# module_autoload is used by the kernel packaging bbclass
module_autoload_atmel_usba_udc = "atmel_usba_udc"
module_autoload_g_serial = "g_serial"
```

After the kernel is built by running the `bitbake virtual/kernel` command, the kernel image will be available inside the `tmp/deploy/image/sama5d3-xplained` directory under the `zImage-sama5d3-xplained.bin` name, which is a symbolic link to the full name file and has a larger name identifier. The kernel image was deployed here from the place where the Linux kernel tasks were executed. The simplest method to discover that place would be to run `bitbake -c devshell virtual/kernel`. A development shell will be available to the user for direct interaction with the Linux kernel source code and access to task scripts. This method is preferred because the developer has access to the same environment as `bitbake`.

A kernel module, on the other hand, has a different kind of behavior if it is not built-in inside the Linux kernel source tree. For the modules that are build outside of the source tree, a new recipe need to be written, that is, a recipe that inherits another `bitbake` class this time called `module.bbclass`. One example of an external Linux kernel module is available inside the `meta-skeleton` layer in the `recipes-kernel/hello-mod` directory:

```
SUMMARY = "Example of how to build an external Linux kernel module"
LICENSE = "GPLv2"
LIC_FILES_CHKSUM = "file://COPYING;md5=12f884d2ae1ff87c09e5b7ccc2c4ca
7e"

inherit module

PR = "r0"
PV = "0.1"

SRC_URI = "file://Makefile \
           file://hello.c \
           file://COPYING \
           "

S = "${WORKDIR}"

# The inherit of module.bbclass will automatically name module
# packages with
# "kernel-module-" prefix as required by the oe-core build
# environment.
```

As mentioned in the example of the Linux kernel external module, the last two lines of each kernel module that is external or internal is packaged with the `kernel-module-` prefix to make sure that when the `IMAGE_INSTALL` variable is available, the value `kernel-modules` are added to all kernel modules available inside the `/lib/modules/<kernel-version>` directory. The kernel module recipe is very similar to any available recipe, the major difference being in the form of the module inherited, as shown in the line `inherit module`.

Inside the Yocto Project, there are multiple commands available to interact with the kernel and kernel module recipes. The simplest command is, of course, `bitbake <recipe-name>`, but for the Linux kernel, there are a number of commands available to make the interaction easier. The most used one is the `bitbake -c menuconfig` virtual/kernel operation, which offers access to the kernel configuration menu.

Besides already known tasks, such as `configure`, `compile`, and `devshell`, that are used mostly in the development process, there are other ones, such as `diffconfig`, which uses the `diffconfig` script available in the Linux kernel `scripts` directory. The difference between the implementation of the Yocto Project and the available script of the Linux kernel is the fact that the former adds the kernel config creation phase. These config fragments are used to add kernel configurations into the `.config` file as part of the automation process.

Summary

In this chapter, you learned about the Linux kernel in general, about its features and methods of interacting with it. There was also information about debugging and porting features. All this was done to make sure that you would get enough information about the whole ecosystem before interacting with it. It is my opinion that if you understand the whole picture first, it will become easier to focus on the more specific things. This is also one of the reasons that the Yocto Project reference was kept toward the end. You were introduced to how a Linux kernel recipe and a Linux kernel external module are defined and used later by a given machine. More information on Linux kernels will also be available in the next chapter, which will gather all the previously presented information and will show you how a developer can interact with a Linux operating system image.

Besides this information, in the next chapter, there will be an explanation about the organization of the root file system and the principles behind it, its content, and device drivers. Busybox is another interesting subject that will be discussed and also the various support for file systems that are available. Since it tends to become larger, information about what a minimal file system should look like will also be presented. Having said this, we shall proceed to the next chapter.

5

The Linux Root Filesystem

In this chapter, you will learn about the root filesystem and its structure. You will also be presented with information about the root filesystem's content, the various device drivers available, and its communication with the Linux kernel. We will slowly make the transition to the Yocto Project and the method used to define the Linux root filesystem's content. The necessary information will be presented to make sure that a user will be also able to customize the `rootfs` filesystem according to its needs.

The special requirements of the root filesystem will be presented. You will be given information on its content, subdirectories, defined purposes, the various filesystem options available, the BusyBox alternative, and also a lot of interesting features.

When interacting with an embedded environment, a lot of developers would start from a minimal root filesystem made available by a distribution provider, such as Debian, and using a cross-toolchain will enhance it with various packages, tools, and utilities. If the number of packages to be added is big, it can be very troublesome work. Starting from scratch would be an even bigger nightmare. Inside the Yocto Project, this job is automatized and there is no need for manual work. The development is started from scratch, and it offers a large number of packages inside the root filesystem to make the work fun and interesting. So, let's move ahead and take a look at this chapter's content to understand more about root filesystems in general.

Interacting with the root filesystem

A root filesystem consists of a directory and file hierarchy. In this file hierarchy, various filesystems can be mounted, revealing the content of a specific storage device. The mounting is done using the `mount` command, and after the operation is done, the mount point is populated with the content available on the storage device. The reverse operation is called `umount` and is used to empty the mount point of its content.

The preceding commands are very useful for the interaction of applications with various files available, regardless of their location and format. For example, the standard form for the `mount` command is `mount -t type device directory`. This command asks the kernel to connect the filesystem from the device that has the `type` format mentioned in the command line, along with the directory mentioned in the same command. The `umount` command needs to be given before removing the device to make sure the kernel caches are written in the storage point.

A root filesystem is available in the root hierarchy, also known as `/`. It is the first available filesystem and also the one on which the `mount` command is not used, since it is mounted directly by the kernel through the `root=` argument. The following are the multiple options to load the root filesystem:

- From the memory
- From the network using NFS
- From a NAND chip
- From an SD card partition
- From a USB partition
- From a hard disk partition

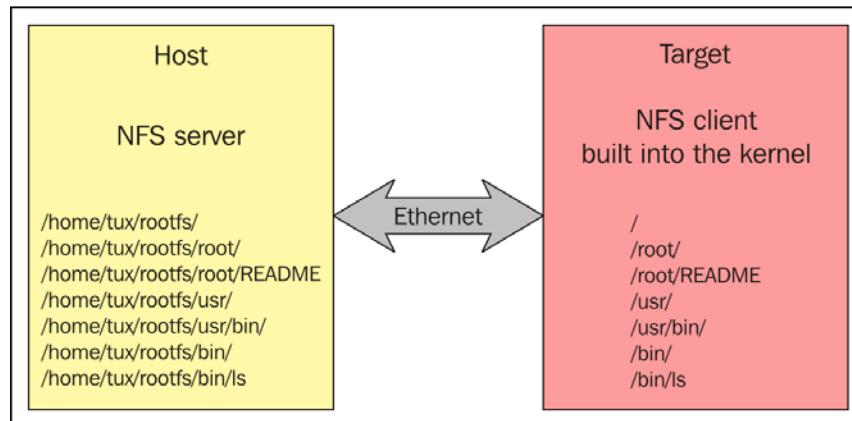
These options are chosen by hardware and system architects. To make use of these, the kernel and bootloader need to be configured accordingly.

Besides the options that require interaction with a board's internal memory or storage devices, one of the most used methods to load the root filesystem is represented by the NFS option, which implies that the root filesystem is available on your local machine and is exported over the network on your target. This option offers the following advantages:

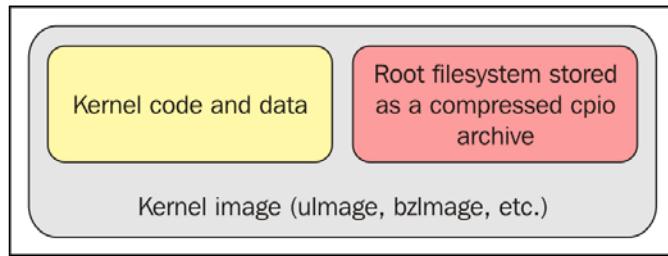
- The size of the root filesystem will not be an issue due to the fact that the storage space on the development machine is much larger than the one available on the target
- The update process is much easier and can be done without rebooting
- Having access to an over the network storage is the best solution for devices with small even nonexistent internal or external storage devices

The downside of the over the network storage is the fact that a sever client architecture is needed. So, for NFS, an NFS server functionality will need to be available on the development machine. For a Ubuntu host, the required configuration involves installing the `nfs-kernel-server` package, `sudo apt-get install nfs-kernel-server`. After the package is installed, the exported directory location needs to be specified and configured. This is done using the `/etc/exports` file; here, configuration lines similar to `/nfs/rootfs <client-IP-address> (rw, no_root_squash, no_subtree_check)` appear, where each line defines a location for the over the network shared locations with the NFS client. After the configuration is finished, the NFS server needs to be restarted in this way: `sudo /etc/init.d/nfs-kernel-server restart`.

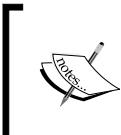
For the client side available on the target, the Linux kernel needs to be configured accordingly to make sure that the NFS support is enabled, and also that an IP address will be available at boot time. This configurations are `CONFIG_NFS_FS=y`, `CONFIG_IP_PNP=y`, and `CONFIG_ROOT_NFS=y`. The kernel also needs to be configured with the `root=/dev/nfs` parameter, the IP address for the target, and the NFS server `nfsroot=192.168.1.110:/nfs/rootfs` information. Here is an example of the communication between the two components:



There is also the possibility of having a root filesystem integrated inside the kernel image, that is, a minimal root filesystem whose purpose is to start the full featured root filesystem. This root filesystem is called `initramfs`. This type of filesystem is very helpful for people interested in fast booting options of smaller root filesystems that only contain a number of useful features and need to be started earlier. It is useful for the fast loading of the system at boot time, but also as an intermediate step before starting the real root filesystem available on one of the available storage locations. The root filesystem is first started after the kernel booting process, so it makes sense for it to be available alongside the Linux kernel, as it resides near the kernel on the RAM memory. The following image explains this:



To create `initramfs`, configurations need to be made available. This happens by defining either the path to the root filesystem directory, the path to a `cpio` archive, or even a text file describing the content of the `initramfs` inside the `CONFIG_INITRAMFS_SOURCE`. When the kernel build starts, the content of `CONFIG_INITRAMFS_SOURCE` will be read and the root filesystem will be integrated inside the kernel image.



More information about the `initramfs` filesystem's options can be found inside the kernel documentations files at `Documentation/filesystems/ramfs-rootfs-initramfs.txt` and `Documentation/early-userspace/README`.

The initial RAM disk or `initrd` is another mechanism of mounting an early root filesystem. It also needs the support enabled inside the Linux kernel and is loaded as a component of the kernel. It contains a small set of executables and directories and represents a transient stage to the full featured root filesystem. It only represents the final stage for embedded devices that do not have a storage device capable of fitting a bigger root filesystem.

On a traditional system, the `initrd` is created using the `mkinitrd` tool, which is, in fact, a shell script that automates the steps necessary for the creation of `initrd`. Here is an example of its functionality:

```
#!/bin/bash

# Housekeeping...
rm -f /tmp/ramdisk.img
rm -f /tmp/ramdisk.img.gz

# Ramdisk Constants
RDSIZE=4000
BLKSIZE=1024

# Create an empty ramdisk image
dd if=/dev/zero of=/tmp/ramdisk.img bs=$BLKSIZE count=$RDSIZE

# Make it an ext2 mountable file system
/sbin/mke2fs -F -m 0 -b $BLKSIZE /tmp/ramdisk.img $RDSIZE

# Mount it so that we can populate
mount /tmp/ramdisk.img /mnt/initrd -t ext2 -o loop=/dev/loop0

# Populate the filesystem (subdirectories)
mkdir /mnt/initrd/bin
mkdir /mnt/initrd/sys
mkdir /mnt/initrd/dev
mkdir /mnt/initrd/proc

# Grab busybox and create the symbolic links
pushd /mnt/initrd/bin
cp /usr/local/src/busybox-1.1.1/busybox .
```

```
ln -s busybox ash
ln -s busybox mount
ln -s busybox echo
ln -s busybox ls
ln -s busybox cat
ln -s busybox ps
ln -s busybox dmesg
ln -s busybox sysctl
popd

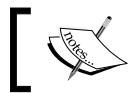
# Grab the necessary dev files
cp -a /dev/console /mnt/initrd/dev
cp -a /dev/ramdisk /mnt/initrd/dev
cp -a /dev/ram0 /mnt/initrd/dev
cp -a /dev/null /mnt/initrd/dev
cp -a /dev/tty1 /mnt/initrd/dev
cp -a /dev/tty2 /mnt/initrd/dev

# Equate sbin with bin
pushd /mnt/initrd
ln -s bin sbin
popd

# Create the init file
cat >> /mnt/initrd/linuxrc << EOF
#!/bin/ash
echo
echo "Simple initrd is active"
echo
mount -t proc /proc /proc
mount -t sysfs none /sys
/bin/ash --login
EOF

chmod +x /mnt/initrd/linuxrc
```

```
# Finish up...
umount /mnt/initrd
gzip -9 /tmp/ramdisk.img
cp /tmp/ramdisk.img.gz /boot/ramdisk.img.gz
```



More information on initrd can be found at Documentation/
initrd.txt.



Using initrd is not as simple as initramfs. In this case, an archive needs to be copied in a similar manner to the one used for the kernel image, and the bootloader needs to pass its location and size to the kernel to make sure that it has started. Therefore, in this case, the bootloader also requires the support of initrd. The central point of the initrd is constituted by the `linuxrc` file, which is the first script started and is usually used for the purpose of offering access to the final stage of the system boot, that is, the real root filesystem. After `linuxrc` finishes the execution, the kernel unmounts it and continues with the real root filesystem.

Delving into the filesystem

No matter what their provenience is, most of the available root filesystems have the same organization of directories, as defined by the **Filesystem Hierarchy Standard (FHS)**, as it is commonly called. This organization is of great help to both developers and users because it not only mentions a directory hierarchy, but also the purpose and content of the directories. The most notable ones are:

- `/bin`: This refers to the location of most programs
- `/sbin`: This refers to the location of system programs
- `/boot`: This refers to the location for boot options, such as the kernel image, kernel config, initrd, system maps, and other information
- `/home`: This refers to the user home directory
- `/root`: This refers to the location of the root user's home location
- `/usr`: This refers to user-specific programs and libraries, and mimics parts of the content of the root filesystem
- `/lib`: This refers to the location of libraries
- `/etc`: This refers to the system-wide configurations
- `/dev`: This refers to the location of device files
- `/media`: This refers to the location of mount points of removable devices
- `/mnt`: This refers to the mount location point of static media

- `/proc`: This refers to the mounting point of the `proc` virtual filesystem
- `/sys`: This refers to the mounting point of the `sysfs` virtual filesystem
- `/tmp`: This refers to the location temporary files
- `/var`: This refers to data files, such as logging data, administrative information, or the location of transient data

The FHS changes over time, but not very much. Most of the previously mentioned directories remain the same for various reasons - the simplest one being the fact that they need to ensure backward compatibility.



The latest available information of the FHS is available at http://refspecs.linuxfoundation.org/FHS_2.3/fhs-2.3.pdf.



The root filesystems are started by the kernel, and it is the last step done by the kernel before it ends the boot phase. Here is the exact code to do this:

```
/*
 * We try each of these until one succeeds.
 *
 * The Bourne shell can be used instead of init if we are
 * trying to recover a really broken machine.
 */
if (execute_command) {
    ret = run_init_process(execute_command);
    if (!ret)
        return 0;
    pr_err("Failed to execute %s (error %d). Attempting
defaults...\n", execute_command, ret);
}
if (!try_to_run_init_process("/sbin/init") ||
    !try_to_run_init_process("/etc/init") ||
    !try_to_run_init_process("/bin/init") ||
    !try_to_run_init_process("/bin/sh"))
    return 0;

panic("No working init found. Try passing init= option to
kernel." "See Linux Documentation/init.txt for guidance.");
```

In this code, it can easily be identified that there are a number of locations used for searching the `init` process that needs to be started before exiting from the Linux kernel boot execution. The `run_init_process()` function is a wrapper around the `execve()` function that will not have a return value if no errors are encountered in the call procedure. The called program overwrites the memory space of the executing process, replacing the calling thread and inheriting its `PID`.

This initialization phase is so old that a similar structure inside the Linux 1.0 version is also available. This represents the user space processing start. If the kernel is not able to execute one of the four preceding functions in the predefined locations, then the kernel will halt and a panic message will be prompted onto the console to issue an alert that no `init` processes can be started. So, the user space processing will not start until the kernel space processing is finished.

For the majority of the available Linux systems, `/sbin/init` is the location where the kernel spawns the `init` process; the same affirmation is also true for the Yocto Project's generated root filesystems. It is the first application run in the user space context, but it isn't the only necessary feature of the root filesystem. There are a couple of dependencies that need to be resolved before running any process inside the root filesystem. There are dependencies used to solve dynamically linked dependencies references that were not solved earlier, and also dependencies that require external configurations. For the first category of dependencies, the `ldd` tool can be used to spot the dynamically linked dependencies, but for the second category, there is no universal solution. For example, for the `init` process, the configuration file is `inittab`, which is available inside the `/etc` directory.

For developers not interested in running another `init` process, this option is available and can be accessed using the kernel command line with the available `init=` parameter, where the path to the executed binary should be made available. This information is also available in the preceding code. The customization of the `init` process is not a method commonly used by developers, but this is because the `init` process is a very flexible one, which makes a number of start up scripts available.

Every process started after `init` uses the parent-child relationship, where `init` acts as the parent for all the processes run in the user space context, and is also the provider of environment parameters. Initially, the `init` process spawns processes according to the information available inside the `/etc/inittab` configuration file, which defines the runlevel notion. A runlevel represents the state of the system and defines the programs and services that have been started. There are eight runlevels available, numbered from 0 to 6, and a special one that is noted as `s`. Their purpose is described here:

Runlevel value	Runlevel purpose
0	It refers to the shutdown and power down command for the whole system
1	It is a single-user administrative mode with a standard login access
2	It is multiuser without a TCP/IP connection
3	It refers to a general purpose multiuser
4	It is defined by the system's owner
5	It refers to graphical interface and TCP/IP connection multiuser systems
6	It refers to a system reboot
s	It is a single user mode that offers access to a minimal root shell

Each runlevel starts and kills a number of services. The services that are started begin with s, and the ones that are killed begin with k. Each service is, in fact, a shell script that defines the behaviour of the provides that it defines.

The /etc/inittab configuration script defines the runlevel and the instructions applied to all of them. For the Yocto Project, the /etc/inittab looks similar to this:

```
# /etc/inittab: init(8) configuration.
# $Id: inittab,v 1.91 2002/01/25 13:35:21 miquels Exp $

# The default runlevel.
id:5:initdefault:

# Boot-time system configuration/initialization script.
# This is run first except when booting in emergency (-b) mode.
si::sysinit:/etc/init.d/rcS

# What to do in single-user mode.
~~:S:wait:/sbin/sulogin

# /etc/init.d executes the S and K scripts upon change
# of runlevel.
#
# Runlevel 0 is halt.
# Runlevel 1 is single-user.
# Runlevels 2-5 are multi-user.
# Runlevel 6 is reboot.
```

```
10:0:wait:/etc/init.d/rc 0
11:1:wait:/etc/init.d/rc 1
12:2:wait:/etc/init.d/rc 2
13:3:wait:/etc/init.d/rc 3
14:4:wait:/etc/init.d/rc 4
15:5:wait:/etc/init.d/rc 5
16:6:wait:/etc/init.d/rc 6
# Normally not reached, but fallthrough in case of emergency.
z6:6:respawn:/sbin/sulogin
S0:12345:respawn:/sbin/getty 115200 ttys0
# /sbin/getty invocations for the runlevels.
#
# The "id" field MUST be the same as the last
# characters of the device (after "tty").
#
# Format:
# <id>:<runlevels>:<action>:<process>
#
1:2345:respawn:/sbin/getty 38400 tty1
```

When the preceding `inittab` file is parsed by the `init`, the first script that is executed is the `si::sysinit:/etc/init.d/rcS` line, identified through the `sysinit` tag. Then, runlevel 5 is entered and the processing of instructions continues until the last level, until a shell is finally spawned using `/sbin/getty` symlink. More information on either `init` or `inittab` can be found by running `man init` or `man inittab` in the console.

The last stage of any Linux system is represented by the power off or shutdown command. It is very important, because if it's not done appropriately, it can affect the system by corrupting data. There are, of course, multiple options to implement the shutdown scheme, but the handiest ones remain in the form of utilities, such as `shutdown`, `halt`, or `reboot`. There is also the possibility to use `init 0` to halt the system, but, in fact, what all of them have in common is the use of the `SIGTERM` and `SIGKILL` signals. `SIGTERM` is used initially to notify you about the decision to shut down the system, to offer the chance to the system to perform necessary actions. After this is done, the `SIGKILL` signal is sent to terminate all the processes.

Device drivers

One of the most important challenges for the Linux system is the access allowed to applications to various hardware devices. Notions, such as virtual memory, kernel space, and user space, do not help in simplifying things, but add another layer of complexity to this information.

A device driver has the sole purpose of isolating hardware devices and kernel data structures from user space applications. A user does not need to know that to write data to a hard disk, he or she will be required to use sectors of various sizes. The user only opens a file to write inside it and close when finished. The device driver is the one that does all the underlying work, such as isolating complexities.

Inside the user space, all the device drivers have associated device nodes, which are, in fact, special files that represent a device. All the device files are located in the `/dev` directory and the interaction with them is done through the `mknod` utility. The device nodes are available under two abstractions:

- **Block devices:** These are composed of fixed size blocks that are usually used when interacting with hard disks, SD cards, USB sticks, and so on
- **Character devices:** These are streams of characters that do not have a size, beginning, or end; they are mostly not in the form of block devices, such as terminals, serial ports, sound card and so on

Each device has a structure that offers information about it:

- Type identifies whether the device node is a character or block
- Major identifies the category for the device
- Minor holds the identifier of the device node

The `mknod` utility that creates the device node uses a triplet of information, such as `mknod /dev/testdev c 234 0`. After the command is executed, a new `/dev/testdev` file appears. It should bind itself to a device driver that is already installed and has already defined its properties. If an open command is issued, the kernel looks for the device driver that registered with the same major number as the device node. The minor number is used for handling multiple devices, or a family of devices, with the same device driver. It is passed to the device driver so that it can use it. There is no standard way to use the minor, but usually, it defines a specific device from a family of the devices that share the same major number.

Using the `mknod` utility requires manual interaction and root privileges, and lets the developer do all the heavy lifting needed to identify the properties of the device node and its device driver correspondent. The latest Linux system offers the possibility to automate this process and to also complete these actions every time devices are detected or disappear. This is done as follows:

- `devfs`: This refers to a device manager that is devised as a filesystem and is also accessible on a kernel space and user space.
- `devtmpfs`: This refers to a virtual filesystem that has been available since the 2.6.32 kernel version release, and is an improvement to `devfs` that is used for boot time optimizations. It only creates device nodes for hardware available on a local system.
- `udev`: This refers to a daemon used on servers and desktop Linux systems. More information on this can be referred to by accesing <https://www.kernel.org/pub/linux/utils/kernel/hotplug/udev/udev.html>. The Yocto Project also uses it as the default device manager.
- `mdev`: This offers a simpler solution then `udev`; it is, in fact, a derivation of `udev`.

Since system objects are also represented as files, it simplifies the method of interaction with them for applications. This would not been possible without the use of device nodes, that are actually files in which normal file interaction functions can be applied, such as `open()`, `read()`, `write()`, and `close()`.

Filesystem options

The root filesystem can be deployed under a very broad form of the filesystem type, and each one does a particular task better than the rest. If some filesystems are optimized for performance, others are better at saving space or even recovering data. Some of the most commonly used and interesting ones will be presented here.

The logical division for a physical device, such as a hard disk or SD card, is called a **partition**. A physical device can have one or more partitions that cover its available storage space. It can be viewed as a logical disk that has a filesystem available for the user's purposes. The management of partitions in Linux is done using the `fdisk` utility. It can be used to `create`, `list`, `destroy`, and other general interactions, with more than 100 partition types. To be more precise, 128 partition types are available on my Ubuntu 14.04 development machine.

One of the most used and well known filesystem partition formats is `ext2`. Also called **second extended filesystem**, it was introduced in 1993 by Rémy Card, a French software developer. It was used as the default filesystem for a large number of Linux distributions, such as Debian and Red Hat Linux, until it was replaced by its younger brothers, `ext3` and `ext4`. It continues to remain the choice of many embedded Linux distributions and flash storage devices.

The `ext2` filesystem splits data into blocks, and the blocks are arranged into block groups. Each block group maintains a copy of a superblock and the descriptor table for that block group. Superblocks are to store configuration information, and hold the information required by the booting process, although there are available multiple copies of it; usually, the first copy that is situated in the first block of the file system is the one used. All the data for a file is usually kept in a single block so that searches can be made faster. Each block group, besides the data it contains, has information about the superblock, descriptor table for the block group, inode bitmap and table information, and the block bitmap. The superblock is the one that holds the information important for the booting process. Its first block is used for the booting process. The last notion presented is in the form of `inodes`, or the index nodes, which represent files and directories by their permission, size, location on disk, and ownership.

There are multiple applications used for interaction with the `ext2` filesystem format. One of them is `mke2fs`, which is used to create an `ext2` filesystem on a `mke2fs /deb/sdb1 -L` partition (`ext2` label partition). The is the `e2fsck` command, which is used to verify the integrity of the filesystem. If no errors are found, these tools give you information about the partition filesystem configuration, `e2fsck /dev/sdb1`. This utility is also able to fix some of the errors that appear after improper utilization of the device, but cannot be used in all scenarios.

`Ext3` is another powerful and well known filesystem. It replaced `ext2` and became one of the most used filesystems on Linux distributions. It is in fact similar to `ext2`; the difference being that it has the possibility to journalize the information available to it. The `ext2` file format can be changed in an `ext3` file format using the `tune2fs -j /dev/sdb1` command. It is basically seen as an extension for the `ext2` filesystem format, one that adds the journaling feature. This happens because it was engineered to be both forward and backward compatible.

Journaling is a method that is used to log all the changes made on a filesystem form by making the recovery functionality possible. There are also other features that `ext3` adds besides the ones that are already mentioned; here, I am referring to the possibility of not checking for consistencies in the filesystem, mostly because journalizing the log can be reversed. Another important feature is that it can be mounted without checking whether the shutdown was performed correctly. This takes place because the system does not need to conduct a consistency check at power down.

`Ext4` is the successor of `ext3`, and was built with the idea of improving the performance and the storage limit in `ext3`. It is also backward compatible with the `ext3` and `ext2` filesystems and also adds a number of features:

- Persistent preallocation: This defines the `fallocate()` system call that can be used to preallocate space, which is most likely in a contiguous form; it is very useful for databases and streaming of media
- Delayed allocation: This is also called **allocate-on-flush**; it is used to delay the allocation blocks from the moment data from the disk is flushed, to reduce fragmentation and increase performance
- Multi block allocation: This is a side effect of delayed allocation because it allows for data buffering and, at the same time, the allocation of multiple blocks.
- Increase subdirectory limit: This the `ext3` has a limit of 32000 subdirectories, the `ext4` does not have this limitation, that is, the number of subdirectories are unlimited
- Checksum for journal: This is used to improve reliability

Journalling Flash Filesystem version 2 (JFFS2) is a filesystem designed for the NAND and NOR flash memory. It was included in the Linux mainline kernel in 2001, the same year as the `ext3` filesystem, although in different months. It was released in November for the Linux version 2.4.15, and the JFFS2 filesystem was released in September with the 2.4.10 kernel release. Since it's especially used to support flash devices, it takes into consideration certain things, such as the need to work with small files, and the fact that these devices have a wear level associated with them, which solves and reduces them by their design. Although JFFS2 is the standard for flash memory, there are also alternatives that try to replace it, such as LogFS, Yet Another Flash File System (YAFFS), and Unsorted Block Image File System (UBIFS).

Besides the previously mentioned filesystems, there are also some pseudo filesystems available, including `proc`, `sysfs`, and `tmpfs`. In the next section, the first two of them will be described, leaving the last one for you to discover by yourself.

The `proc` filesystem is a virtual filesystem available from the first version of Linux. It was defined to allow a kernel to offer information to the user about the processes that are run, but over time, it has evolved and is now able to not only offer statistics about processes that are run, but also offer the possibility to adjust various parameters regarding the management of memory, processes, interrupts, and so on.

With the passing of time, the `proc` virtual filesystem became a necessity for Linux system users since it gathered a very large number of user space functionalities. Commands, such as `top`, `ps`, and `mount` would not work without it. For example, the `mount` example given without a parameter will present `proc` mounted on the `/proc` in the form of `proc` on `/proc` type `proc` (`rw, noexec, nosuid, nodev`). This takes place since it is necessary to have `proc` mounted on the root filesystem on par with directories, such as `/etc`, `/home`, and others that are used as the destination of the `/proc` filesystem. To mount the `proc` filesystem, the `mount -t proc nodev/proc` mount command that is similar to the other filesystems available is used. More information on this can be found inside the kernel sources documentation at `Documentation/filesystems/proc.txt`.

The `proc` filesystem has the following structure:

- For each running process, there is an available directory inside `/proc/<pid>`. It contains information about opened files, used memory, CPU usage, and other process-specific information.
- Information on general devices is available inside `/proc/devices`, `/proc/interrupts`, `/proc/ioports`, and `/proc/iomem`.
- The kernel command line is available inside `/proc/cmdline`.
- Files used to change kernel parameters are available inside `/proc/sys`. More information is also available inside `Documentation/sysctl`.

The `sysfs` filesystem is used for the representation of physical devices. It is available since the introduction of the 2.6 Linux kernel versions, and offers the possibility of representing physical devices as kernel objects and associate device drivers with corresponding devices. It is very useful for tools, such as `udev` and other device managers.

The `sysfs` directory structure has a subdirectory for every major system device class, and it also has a system buses subdirectory. There is also `systool` that can be used to browse the `sysfs` directory structure. Similar to the `proc` filesystem, `systool` also can also be visible if the `sysfs` on `/sys` type `sysfs` (`rw, noexec, nosuid, nodev`) mount command is offered on the console. It can be mounted using the `mount -t sysfs nodev /sys` command.



More information on available filesystems can be found at http://en.wikipedia.org/wiki/List_of_file_systems.



Understanding BusyBox

BusyBox was developed by Bruce Perens in 1999 with the purpose of integrating available Linux tools in a single executable. It has been used with great success as a replacement for a great number of Linux command line utilities. Due to this, and the fact that it is able to fit inside small embedded Linux distributions, it has gained a lot of popularity in the embedded environment. It provides utilities from file interactions, such as `cp`, `mkdir`, `touch`, `ls`, and `cat`, as well as general utilities, such as `dmesg`, `kill`, `fdisk`, `mount`, `umount`, and many others.

Not only is it very easy to configure and compile, but it is also very easy to use. The fact that it is very modular and offers a high degree of configuration makes it the perfect choice to use. It may not include all the commands available in a full-blown Linux distribution available on your host PC, but the ones that it does are more than enough. Also, these commands are just simpler versions of the full-blown ones used at implementation level, and are all integrated in one single executable available in `/bin/busybox` as symbolic links of this executable.

A developer interaction with the BusyBox source code package is very simple: just configure, compile, and install it, and there you have it. Here are some detailed steps to explain the following:

- Run the configuration tool and chose the features you want to make available
- Execute `make dep` to construct the dependencies tree
- Build the package using the `make` command



Install the executable and symbolic links on the target. People who are interested in interacting with the tool on their workstations should note that if the tool is installed for the host system, then the installation should be done in a location that does not overwrite any of the utilities and start up scripts available to the host.

The configuration of the BusyBox package also has a `menuconfig` option available, similar to the one available for the kernel and U-Boot, that is, `make menuconfig`. It is used to show a text menu that can be used for faster configuration and configuration searches. For this menu to be available, first the `ncurses` package needs to be available on the system that calls the `make menuconfig` command.

At the end of the process, the BusyBox executable is available. If it's called without arguments, it will present an output very similar to this:

```
Usage: busybox [function] [arguments]...
or: [function] [arguments]...

BusyBox is a multi-call binary that combines many common Unix
utilities into a single executable. Most people will create a
link to busybox for each function they wish to use and BusyBox
will act like whatever it was invoked as!

Currently defined functions:
[, [], arping, ash, awk, basename, bunzip2, busybox, bzcat, cat,
chgrp, chmod, chown, chroot, clear, cp, crond, crontab, cut,
date,
dd, df, dirname, dmesg, du, echo, egrep, env, expr, false, fgrep,
find, free, grep, gunzip, gzip, halt, head, hexdump, hostid,
hostname,
id, ifconfig, init, insmod, ipcalc, ipkg, kill, killall,
killall5,
klogd, length, ln, lock, logger, logread, ls, lsmod, md5sum,
mesg,
mkdir, mkfifo, mktemp, more, mount, mv, nc, "netmsg", netstat,
nslookup, passwd, pidof, ping, pivot_root, poweroff, printf, ps,
pwd, rdate, reboot, reset, rm, rmdir, rmmmod, route, sed, seq,
sh, sleep, sort, strings, switch_root, sync, sysctl, syslogd,
tail, tar, tee, telnet, test, time, top, touch, tr, traceroute,
true, udhcpc, umount, uname, uniq, uptime, vi, wc, wget, which,
xargs, yes, zcat
```

It presents the list of the utilities enabled in the configuration stage. To invoke one of the preceding utilities, there are two options. The first option requires the use of the BusyBox binary and the number of utilities called, which are represented as `./busybox 1s`, while the second option involves the use of the symbolic link already available in directories, such as `/bin`, `/sbin`, `/usr/bin`, and so on.

Besides the utilities that are already available, BusyBox also offers implementation alternatives for the `init` program. In this case, the `init` does not know about a runlevel, and all its configurations available inside the `/etc/inittab` file. Another factor that differentiates it from the standard `/etc/inittab` file is the fact that this one also has its special syntax. For more information, `examples/inittab` available inside BusyBox can be consulted. There are also other tools and utilities implemented inside the BusyBox package, such as a lightweight version for `vi`, but I will let you discover them for yourself.

Minimal root filesystem

Now that all the information relating to the `root` filesystem has been presented to you, it would be good exercise to describe the must-have components of the minimal root filesystem. This would not only help you to understand the `rootfs` structure and its dependencies better, but also help with requirements needed for boot time and the size optimization of the `root` filesystem.

The starting point to describe the components is `/sbin/init`; here, by using the `ldd` command, the runtime dependencies can be found. For the Yocto Project, the `ldd /sbin/init` command returns:

```
linux-gate.so.1 (0xb7785000)
libc.so.6 => /lib/libc.so.6 (0x4273b000)
/lib/ld-linux.so.2 (0x42716000)
```

From this information, the `/lib` directory structure is defined. Its minimal form is:

```
lib
| -- ld-2.3.2.so
| -- ld-linux.so.2 -> ld-2.3.2.so
| -- libc-2.3.2.so
'-- libc.so.6 -> libc-2.3.2.so
```

The following symbolic links to ensure backward compatibility and version immunity for the libraries. The `linux-gate.so.1` file in the preceding code is a **virtual dynamically linked shared object** (vDSO), exposed by the kernel at a well established location. The address where it can be found varies from one machine architecture to another.

After this, `init` and its runlevel must be defined. The minimal form for this is available inside the BusyBox package, so it will also be available inside the `/bin` directory. Alongside it, a symbolic link for shell interaction is necessary, so this is how the minimal for the `bin` directory will look:

```
bin
| -- busybox
'-- sh -> busybox
```

Next, the runlevel needs to be defined. Only one is used in the minimal `root` filesystem, not because it is a strict requirement, but due to the fact that it can suppress some BusyBox warnings. This is how the `/etc` directory will look:

```
etc
'-- init.d
    '-- rcs
```

At the end, the console device needs to be available to the user for input and output operations, so the last piece of the `root` filesystem is inside the `/dev` directory:

```
dev
'-- console
```

Having mentioned all of this, the minimal `root` filesystem seems to have only five directories and eight files. Its minimal size is below 2 MB and around 80 percent of its size is due to the C library package. It is also possible to minimize its size by using the Library Optimizer Tool. You can find more information on this at <http://libraryopt.sourceforge.net/>.

The Yocto Project

Moving to the Yocto Project, we can take a look at the core-image-minimal to identify its content and minimal requirements, as defined inside the Yocto Project. The `core-image-minimal.bb` image is available inside the `meta/recipes-core/images` directory, and this is how it looks:

```
SUMMARY = "A small image just capable of allowing a device to boot."

IMAGE_INSTALL = "packagegroup-core-boot ${ROOTFS_PKGMANAGE_BOOTSTRAP}
${CORE_IMAGE_EXTRA_INSTALL} ldd"

IMAGE_LINGUAS = " "

LICENSE = "MIT"

inherit core-image

IMAGE_ROOTFS_SIZE ?= "8192"
```

You can see here that this is similar to any other recipe. The image defines the `LICENSE` field and inherits a `bbclass` file, which defines its tasks. A short summary is used to describe it, and it is very different from normal package recipes. It does not have `LIC_FILES_CHKSUM` to check for licenses or a `SRC_URI` field, mostly because it does not need them. In return, the file defines the exact packages that should be contained in the `root` filesystem, and a number of them are grouped inside `packagegroup` for easier handling. Also, the `core-image` `bbclass` file defines a number of other tasks, such as `do_rootfs`, which is only specific for image recipes.

Constructing a `root` filesystem is not an easy task for anyone, but Yocto does it with a bit more success. It starts from the `base-files` recipe that is used to lay down the directory structure according to the **Filesystem Hierarchy Standard (FHS)**, and, along with it, a number of other recipes are placed. This information is available inside the `./meta/recipes-core/packagegroups/packagegroup-core-boot.bb` recipe. As can be seen in the previous example, it also inherits a different kind of class, such as `packagegroup.bbclass`, which is a requirement for all the package groups available. However, the most important factor is that it clearly defines the packages that constitute `packagegroup`. In our case, the `core boot` package group contains packages, such as `base-files`, `base-passwd` (which contains the base system master password and group files), `udev`, `busybox`, and `sysvinit` (a System V similar to `init`).

As can be seen in the previously shown file, the BusyBox package is a core component of the Yocto Project's generated distributions. Although information was available about the fact that BusyBox can offer an init alternative, the default Yocto generated distributions do not use this. Instead, they choose to move to the System V-like init, which is similar to the one available for Debian-based distributions. Nevertheless, a number of shell interaction tools are made available through the BusyBox recipe available inside the `meta/recipes-core/busybox` location. For users interested in enhancing or removing some of features made available by the `busybox` package, the same concepts that are available for the Linux kernel configuration are used. The `busybox` package uses a `defconfig` file on which a number of configuration fragments are applied. These fragments can add or remove features and, in the end, the final configuration file is obtained. This identifies the final features available inside the root filesystem.

Inside the Yocto Project, it is possible to minimize the size of the root filesystem by using the `poky-tiny.conf` distribution policies, which are available inside the `meta-yocto/conf/distro` directory. When they're used, these policies reduce not only the boot size, but the boot time as well. The simplest example for this is available using the `qemux86` machine. Here, changes are visible, but they are somewhat different from the ones already mentioned in the *Minimal root filesystem* section. The purpose of the minimization work done on `qemux86` was done around the `core-image-minimal` image. Its goals is to reduce the size to under 4 MB of the resulting `rootfs` and the boot time to under 2 seconds.

Now, moving to the selected Atmel SAMA5D3 Xplained machine, another `rootfs` is generated and its content is quite big. Not only has it included the `packagegroup-core-boot.bb` package group, but other package groups and separate packages are also included. One such example is the `atmel-xplained-demo-image.bb` image available inside the `meta-atmel` layer in the `recipes-core/images` directory:

```
DESCRIPTION = "An image for network and communication."
LICENSE = "MIT"
PR = "r1"

require atmel-demo-image.inc

IMAGE_INSTALL += "\\
    packagegroup-base-3g \
    packagegroup-base-usbhost \
    "
```

Inside this image, there is also another more generic image definition that is inherited. Here, I am referring to the `atmel-demo-image.inc` file, and when opened, you can see that it contains the core of all the `meta-atmel` layer images. Of course, if all the available packages are not enough, a developer could decide to add their own. There has two possibilities in front of a developer: to create a new image, or to add packages to an already available one. The end result is built using the `bitbake atmel-xplained-demo-image` command. The output is available in various forms, and they are highly dependent on the requirements of the defined machine. At the end of the build procedure, the output will be used to boot the root filesystem on the actual board.

Summary

In this chapter, you have learned about the Linux `rootfs` in general, and also about the communication with the organization of the Linux kernel, Linux `rootfs`, its principles, content, and device drivers. Since communication tends to become larger over time, information about how a minimal filesystem should look was also presented to you.

Besides this information, in the next chapter, you will be given an overview of the available components of the Yocto Project, since most of them are outside Poky. You will also be introduced to, and given a brief gist of, each component. After this chapter, a bunch of them will be presented to you and elaborated on.

6

Components of the Yocto Project

In this chapter, you will be given a short introduction to a number of components from the ecosystem of the Yocto Project. This chapter is meant to introduce all of them so that in subsequent chapters they can be presented more elaborately. It also tries to direct readers toward extra readings. For each presented tool, feature, or interesting fact, links are offered to help interested readers search for their own answers to the questions in this book and those that this chapter does not cover.

This chapter is full of guidance and relevant examples for an embedded development process that involves specific Yocto Project tools. The selection of the tools was done in a purely subjective manner. Only the tools that are considered helpful in the development process have been selected. We also considered the fact that some of them could offer new insights into the embedded world and the development for embedded systems in general.

Poky

Poky represents the reference build system for the metadata and tools of the Yocto Project, which are used as starting points for anyone interested in interacting with the Yocto Project. It is platform-independent and provides the tools and mechanisms to build and customize the end result, which is in fact a Linux software stack. Poky is used as the central piece of interaction with the Yocto Project.

When working with the Yocto Project as a developer, it is very important to have information about mailing lists and an **Internet Relay Chat (IRC)** channel. Also, Project Bugzilla can be a source of inspiration in terms of a list of available bugs and features. All of these elements would need a short introduction, so the best starting point would be the Yocto Project Bugzilla. It represents a bug tracking application for the users of the Yocto Project and is the place where problems are reported. The next component is represented by the available channels of IRC. There are two available components on a freenode, one used for Poky and the other for discussions related to the Yocto Project, such as **#poky** and **#yocto**, respectively. The third element is represented by the Yocto Project mailing lists, which are used to subscribe to these mailing lists of the Yocto Project:

- <http://lists.yoctoproject.org/listinfo/yocto>: This refers to the mailing list where the Yocto Project discussions take place
- <http://lists.yoctoproject.org/listinfo/poky>: This refers to the mailing list where discussions regarding the Poky build of the Yocto Project system take place
- <http://lists.yoctoproject.org/listinfo/yocto-announce>: This refers to the mailing list where official announcements of the Yocto Project are made, as well as where milestones of the Yocto Project are presented

With the help of <http://lists.yoctoproject.org/listinfo>, more information can be gathered regarding general and project-specific mailing lists. It contains a list of all the mailing lists available at <https://www.yoctoproject.org/tools-resources/community/mailing-lists>.

In order to initiate development using the Yocto Project in general, and Poky in particular, you should not only use the previously mentioned components; some information regarding these tools should also be made available. A very good explanation of the Yocto Project is available on their documentation page at <https://www.yoctoproject.org/documentation>. Those of you interested in reading a shorter introduction, it may be worth checking out the *Embedded Linux Development with Yocto Project*, Otávio Salvador and Daiane Angolini, by Packt Publishing.

To use the Yocto Project, a number of specific requirements are needed:

- **A host system:** Let's assume that this is a Linux-based host system. However, it is not just any host system; Yocto has certain requirements. The supported operating systems are available inside the `poky.conf` file, available inside directory `meta-yocto/conf/distro`. The supported operating systems are defined in the `SANITY_TESTED_DISTROS` variable, and a few of these systems are as follows:

- Ubuntu-12.04
 - Ubuntu-13.10
 - Ubuntu-14.04
 - Fedora-19
 - Fedora-20
 - CentOS-6.4
 - CentOS-6.5
 - Debian-7.0
 - Debian-7.1
 - Debian-7.2
 - Debian-7.3
 - Debian-7.4
 - Debian-7.5
 - Debian-7.6
 - SUSE-LINUX-12.2
 - openSUSE-project-12.3
 - openSUSE-project-13.1
- **Required packages:** This contains a list of the minimum requirements for the packages available on the host system, besides the ones already available. Of course, this is different from one host system to another and the systems vary according to their purposes. However, for the Ubuntu host, we need the following requirements:
 - **Essentials:** This refers to `sudo apt-get install gawk wget git-core diffstat unzip texinfo gcc-multilib build-essential chrpath socat`
 - **Graphical and Eclipse Plug-in extras:** This refers to `sudo apt-get install libSDL1.2-dev xterm`
 - **Documentation:** This refers to `sudo apt-get install make xsltproc docbook-utils fop dblatex xmlto`
 - **ADT Installer Extras:** This refers to `sudo apt-get install autoconf automake libtool libglib2.0-dev`

- **Yocto Project release:** Before starting any work, one of the available Poky releases should be chosen. This book is based on the dizzy branch, which is the Poky 1.7 version, but a developer can choose whatever fits him or her best. Of course, since the interaction with the project is done using the git versioning system, the user will first need to clone the Poky repository, and any contributions to the project should be submitted as a patch to the open source community. There is also a possibility of getting a tar archive, but this method has some limitations due to the fact that any changes done on the source are harder to trace, and it also limits the interaction with the community involved in the project.

There are other extra optional requirements that should be taken care of if special requirements are needed, as follows:

- **Custom Yocto Project kernel interaction:** If a developer decides that the kernel source Yocto Projects are maintained and are not suitable for their needs, they could get one of the local copies of the Yocto Project supported by kernel versions, available at <http://git.yoctoproject.org/cgit.cgi> under the Yocto Linux Kernel section, and modify it according to their needs. These changes, of course, along with the rest of the kernel sources, will need to reside in a separate repository, preferably git, and it will be introduced to the Yocto world through a kernel recipe.
- **The meta-yocto-kernel-extras git repository:** Here the metadata needed is gathered when building and modifying kernel images. It contains a bunch of bbappend files that can be edited to indicate to the local that the source code has changed, which is a more efficient method to use when you are working on the development of features of the Linux kernel. It is available under the **Yocto Metadata Layers** section at <http://git.yoctoproject.org/cgit.cgi>.
- **Supported Board Support Packages (BSPs):** There are a large number of BSP layers that are available and supported by the Yocto Project. The naming of each BSP layer is very simple, `meta-<bsp-name>`, and can be found at <http://git.yoctoproject.org/cgit.cgi> under the **Yocto Metadata Layers** section. Each BSP layer is, in fact, a collection of recipes that define the behavior and minimum requirements offered by the BSP provider. More information regarding the development of BSP can be found at <http://www.yoctoproject.org/docs/1.7/dev-manual/dev-manual.html#developing-a-board-support-package-bsp>.

- **Eclipse Yocto Plug-ins:** For developers who are interested in writing applications, an Eclipse **Integrated Development Environment (IDE)** is available with Yocto-specific plug-ins. You can find more information on this at <http://www.yoctoproject.org/docs/1.7/dev-manual/dev-manual.html#setting-up-the-eclipse-ide>.

The development process inside the Yocto Project has many meanings. It can refer to the various bugs and features that are available inside the Yocto Project Bugzilla. The developer can assign one of them to his or her account and solve it. Various recipes can be upgraded, and this process also requires the developer's involvement; new features can also be added and various recipes need to be written by developers. All these tasks need to have a well defined process in place that also involves git interaction.

To send changes added in the recipes back into the community, the available create-pull-request and send-pull request scripts can be used. These scripts are available inside the poky repository in the scripts directory. Also, in this section, there are also a bunch of other interesting scripts available, such as the `create-recipe` script, and others that I will let you discover on your own. The other preferred method to send the changes upstream would be to use the manual method, which involves interaction with git commands, such as `git add`, `git commit -s`, `git format-patch`, `git send-email`, and others.

Before moving on to describe the other components presented in this chapter, a review of the existing Yocto Project development models will be made. This process involves these tools made available by the Yocto Project:

- **System development:** This covers the development of the BSP, kernel development, and its configurations. Each of them has a section in the Yocto Project documentation describing respective development processes, as shown at <http://www.yoctoproject.org/docs/1.7/bsp-guide/bsp-guide.html#creating-a-new-bsp-layer-using-the-yocto-bsp-script> and <http://www.yoctoproject.org/docs/1.7/kernel-dev/kernel-dev.html>.
- **User application development:** This covers the development of applications for a targeted hardware device. The information regarding the necessary setup for the application development on the host system is available at <http://www.yoctoproject.org/docs/1.7/adt-manual/adt-manual.html>. This component will also be discussed in the *Eclipse ADT Plug-ins* section of this chapter.

- **Temporary modification of source code:** This covers the temporary modifications that appear in the development process. This involves the solution for various implementation problems that are available in a project's source code. After the problem is solved, the changes need to be available upstream and applied accordingly.
- **Development of a Hob image:** The Hob build system can be used for operating and customizing system images. It is a graphical interface developed in Python as a more efficient interface with the Bitbake build system.
- **Devshell development:** This is a method of development that uses the exact environment of the Bitbake build system's tasks. It is one of the most efficient methods used for debugging or package editing. It is also one of the quickest ways to set up the build environment when writing various components of a project.

For operating systems where the provided components are too old to satisfy the requirements of the Yocto Project, a buildtools toolchain is recommended for providing the required versions of the software. There are two methods used for installing a buildtools tarball. The first method implies the use of an already available prebuilt tarball, and the second one involves building it using the Bitbake build system. More information about this option can be found in the subsections under the **Required Git, tar, and Python Versions** section of the Yocto documentation mega manual available at <http://www.yoctoproject.org/docs/1.7/mega-manual/mega-manual.html#required-git-tar-and-python-versions>.

Eclipse ADT plug-ins

The **Application Development Toolkit**, also called ADT, provides a cross-development platform suitable for custom build and user-targeted applications. It is comprised of the following elements:

- **A cross-toolchain:** It is associated with the sysroot, both of them being automatically generated using Bitbake, and the target-specific metadata is made available by the target hardware supplier.
- **The Quick Emulator environment (Qemu):** It is used to simulate the target hardware.
- **User-space tools:** It improves the overall experience of development of an application
- **Eclipse IDE:** It contains Yocto Project-specific plug-ins

In this section, each of the preceding elements will be discussed, and we will start with the cross-development toolchain. It consists of a cross-linker, cross-debugger, and a cross-compiler that are used for the application development of a target. It also needs the associated target sysroot because the necessary headers and libraries are required when building an application that will run on the target device. The generated sysroot is obtained from the same configuration that generates the root filesystem; this refers to the *image* recipe.

The toolchain can be generated using multiple methods. The most common one is to download the toolchain from <http://downloads.yoctoproject.org/releases/yocto/yocto-1.7/toolchain/>, and get the appropriate toolchain installer for your host and target. One such example is the `poky-glibc-x86_64-core-image-sato-armv7a-vfp-neon-toolchain-1.7.sh` script, which when executed will install the toolchain in the default location of the `/opt/poky/1.7/` directory. This location can be changed if proper arguments are offered in the script before starting the execution of the script.

Another method I prefer to use when generating a toolchain involves the use of the Bitbake build system. Here, I am referring to `meta-ide-support`. When running `bitbake meta-ide-support`, the cross-toolchain is generated and it populates the build directory. After this task is finished, the same result is obtained as in the previously mentioned solution, but in this case, a build directory that is already available is used. The only remaining task for both solutions would be to set up the environment using the script that contains the `environment-setup` string and start using it.

The Qemu emulator offers the possibility to simulate one hardware device when this one is not available. There are multiple ways of making it available in the development process:

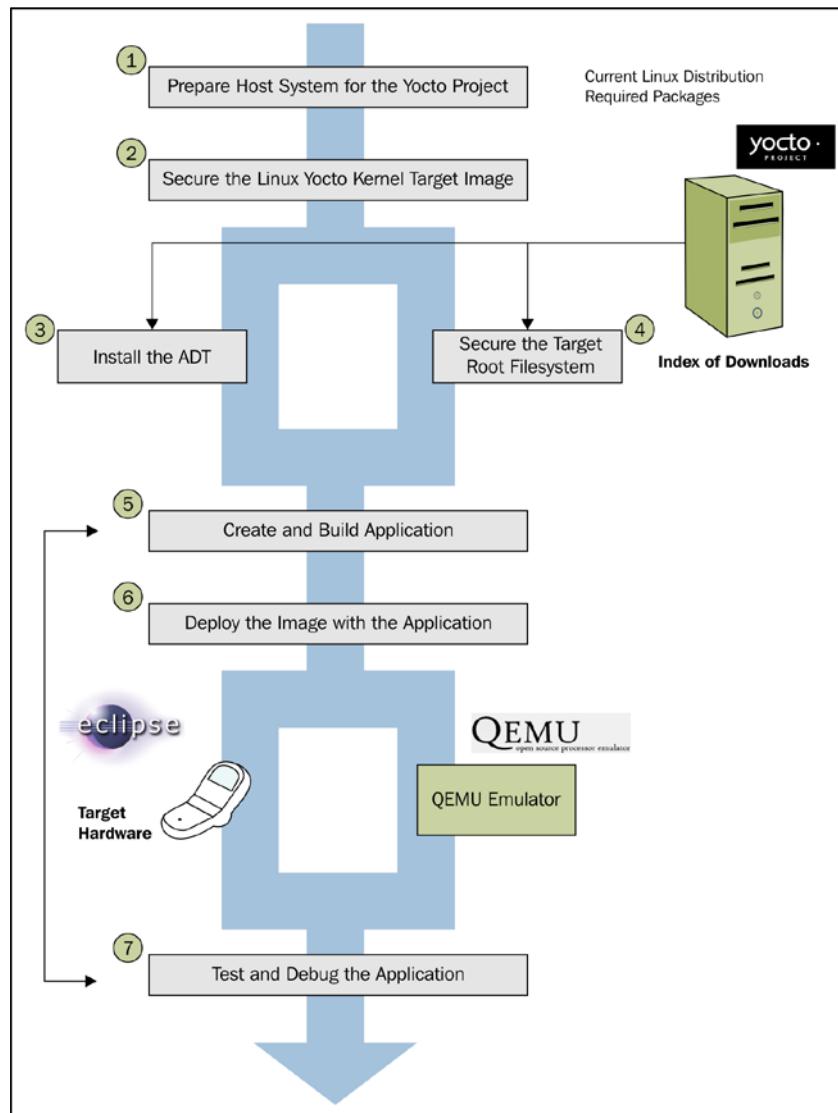
- Install the ADT using the `adt-installer` generated script. One of the steps available in this script offers the possibility to enable or disable the use of Qemu in the development process.
- A Yocto Project release is downloaded and in the development process, the environment is set up by default. Then, the Qemu is installed and available for use.
- A git clone of the Poky repository is created and the environment is set up. In this case, the Qemu is installed and available also.
- The `cross-toolchain` tarball was downloaded, installed, and the environment was set up. This also, by default, enables the use of Qemu and installs it for later use.

The user-space tools are included into the distribution and are used during the development process. They are very common on a Linux platform and can include the following:

- **Perf:** It is a Linux performance counter that measures certain hardware and software events. More information about this is available at <https://perf.wiki.kernel.org/>, and also on the profiling and tracing manual of Yocto, where a whole section is devoted to this tool.
- **PowerTop:** It is a power measurement tool that is used to determine the amount of power a software consumes. More information about it is available at <https://01.org/powertop/>.
- **LatencyTop:** It is a similar tool to PowerTop, the difference being that this one focuses on the latency measurement from audio skips and stutters on the desktop to server overload; it has measurement for these kind of scenarios and answers for the latency problems. Although it seems that no commit has been done inside this project since 2009, it is still used today due to the fact that it is very useful.
- **OProfile:** It represents a system-wide profiler for the Linux ecosystem with a low overhead. More information about it is available at <http://oprofile.sourceforge.net/about/>. It also has a section available in the profiling and tracing manual of Yocto.
- **SystemTap:** It offers information on the infrastructure of a running Linux system, as well as the performance and functional problems of the system. It is not available though as an Eclipse extension, but only as a tool inside the Linux distribution. More information about it can be found at <http://sourceware.org/systemtap>. It also has a section defined in the profiling and tracing manual of Yocto.
- **Lttng-ust:** It is the user-space tracer for the Lttng project and offers information related to user-space activities. More information is available at <http://lttng.org/>.

The last element of the ADT platform is represented by the Eclipse IDE. It is, in fact, the most popular development environment, and it offers full support for the development of the Yocto Project. With the installation of the Yocto Project Eclipse Plug-ins into the Eclipse IDE, the Yocto Project experience is complete. These plugins offer the possibility to cross-compile, develop, deploy, and execute the resultant binary in a Qemu emulated environment. Activities, such as cross-debugging, tracing, remote profiling, and power data collection, are also possible. More information about the activities that appear related to working with Eclipse Plug-ins for the Yocto Project can be found at <http://www.yoctoproject.org/docs/1.7/mega-manual/mega-manual.html#adt-eclipse>.

To better understand the workflow of the application development of the ADT toolkit platform and Eclipse, an overview of the whole process is available in the following image:



The application development process can also be done with other tools that are different from the ones already presented. However, all these options involve the use of a Yocto Project component, most notably the Poby reference system. Therefore, ADT is the suggested, tested, and recommended option by the open source community.

Hob and Toaster

The project – **Hob** – represents a graphical user interface for the Bitbake build system. Its purpose was to simplify the interaction with the Yocto Project and create a leaner learning curve for the project, allowing users to perform daily tasks in a simpler manner. Its primary focus was the generation of a Linux operating system image. With time, it evolved and can now be considered a tool suitable for both experienced and nonexperienced users. Although I mostly prefer using the command line interaction, this statement does not hold true for all Yocto Project users.

It might seem, though, that Hob development stopped with the release of Daisy 1.6. The development activity somewhat moved to the new project – **Toaster** –, which will be explained shortly; the Hob project is still in use today and its functionalities should be mentioned. So, the current available version of Hob is able to do the following:

- Customize an available base image recipe
- Create a completely customized image
- Build any given image
- Run an image using Qemu
- Deploy an image on a USB disk for the purpose of live-booting it on a target

The Hob project can be started in the same way that Bitbake is executed. After the environment sources and the build directory are created, the `hob` command can be called and the graphical interface will appear for the user. The disadvantage of this is that this tool does not substitute the command-line interaction. If new recipes need to be created, then this tool will not be able to provide any help with the task.

The next project is called Toaster. It is an application programming interface and also a web interface that the Yocto Project builds. In its current state, it is only able to gather and present information relevant to a build process through a web browser. These are some of its functionalities:

- Visibility for the executed and reused tasks during the build process
- Visibility for build components, such as recipes and packages of an image - this is done in a manner similar to Hob
- Offering information about recipes, such as dependencies, licenses, and so on
- Offering performance-related information, such as disk I/O , CPU usage, and so on
- Presenting errors, warnings, and trace reports for the purpose of debugging

Although it might not seem much, this project promises to offer the possibility to build and customize builds the same way that Hob did, along with many other goodies. You can find useful information about this tool at: <https://wiki.yoctoproject.org/wiki/Toaster>.

Autobuilder

Autobuilder is a project that facilitates the build test automation and conducts quality assurance. Through this internal project, the Yocto community tries to set a path on which embedded developers are able to publish their QA tests and testing plans, develop new tools for automatic testing, continuous integration, and develop QA procedures to demonstrate and show them for the benefit of all involved parties.

These points are already achieved by a project that publishes its current status using this Autobuilder platform, which is available at <http://autobuilder.yoctoproject.org/>. This link is accessible to everyone and testing is performed on all the changes related to the Yocto Project, as well as nightly builds for all supported hardware platforms. Although started from the Buildbot project, from which it borrowed components for continuous integration, this project promises to move forward and offer the possibility of performing runtime testing and other must-have functionalities.

You can find some useful information about this project at: <https://wiki.yoctoproject.org/wiki/AutoBuilder> and <https://wiki.yoctoproject.org/wiki/QA>, which offers access to the QA procedures done for every release, as well as some extra information.

Lava

The Lava project is not an internal work of the Yocto Project, but is, in fact, a project developed by Linaro, which is an automated validation architecture aimed towards testing the deployments of Linux systems on devices. Although its primary focus is the ARM architecture, the fact that it is open source does not make it a disincentive. Its actual name is **Linaro Automation and Validation Architecture (LAVA)**.

This project offers the possibility of deploying an operating system on a hardware or virtual platform, defining, tests, and performing them on the project. The tests can be of various complexities, they can be combined into bigger and more conclusive tests, and the results are tracked in time, after which the resulting data is exported for analysis.

This is developed with the idea of a continuous evolving architecture that allows test performing along with automation and quality control. At the same time, it offers validation for gathered data. Tests can be anything from compiling a boot test to a change on the kernel scheduler that may or may not have reduced power consumption.

Although it is still young, this project has gained quite an audience, so some investigation into the project would not hurt anyone.



The LAVA manual is available at <https://validation.linaro.org/static/docs/>



Wic

Wic is more of a feature than a project per se. It is the least documented, and if a search is conducted for it, you may find no results. I have decided to mention it here because in the development process, some special requirements could appear, such as generating a custom root filesystem from available packages (such as .deb, .rpm, or .ipk). This job is the one that is best suited for the wic tool.

This tool tries to solve some special requirements from devices or bootloaders, such as special formatting or the partitioning of the root filesystem. It is a highly customized tool that offers the possibility of extending its features. It has been developed from another tool called **oeic**, which was used to create a certain proprietary formatted image for hardware and was imported into the Yocto Project to serve a broader purposes for developers who did not wanted to touch recipes and had already packaged sources, or required special formatting for their deliverable Linux image.

Unfortunately, there is no documentation available for this tool, but I can direct those who are interested to its location on the Yocto Project. It resides in the Poky repository in the scripts directory under the name of wic. Wic can be used as any script, and it provides a help interface where you can seek more information. Also, its functionalities will be presented in an extended manner in the coming chapters.

A list with all the available projects developed around the Yocto Project can be found at <https://www.yoctoproject.org/tools-resources/projects>. Some of the projects available there were not discussed in the context of this chapter, but I will let you discover each one of them. There are also other external projects that did not make the list. I encourage you to find out and learn about them on your own.

Summary

In this chapter, you were presented with the elements that will be discussed next in this book. In the following chapter, each of the previously mentioned sections will be presented in various chapters, and the information will be presented in-depth and in a more applied manner.

In the next chapter, the previously mentioned process will start with the Application Development Toolkit platform. It will be explained with the steps necessary for the setup of the platform, and some usage scenarios will also be introduced to you. These involve cross-development, debugging using Qemu, and the interaction between specific tools.

7

ADT Eclipse Plug-ins

In this chapter, you will be presented with a new perspective of the available tool in the Yocto Project. This chapter marks the beginning of the introduction to various tools available in the Yocto Project ecosystem, tools that are very useful and different from the Poky reference system. In this chapter, a short presentation to the **Application Development Environment (ADE)** is presented with emphasis on the Eclipse project and the Yocto Project's added plug-ins. A number of the plug-ins are shown along with their configurations and use cases.

A broader view of the **Application Development Toolkit (ADT)** will also be shown to you. This project's main objective is to offer a software stack that is able to develop, compile, run, debug, and profile software applications. It tries to do this without requiring extra learning from the developer's point of view. Its learning curve is very low, taking into consideration the fact that Eclipse is one of the most used **Integrated Development Environment (IDEs)**, and over time, it has become very user-friendly, stable, and dependable. The ADT user experience is very similar to the one that any Eclipse or non-Eclipse user has when they use an Eclipse IDE. The available plug-ins try to make this experience as similar as possible so that development is similar to any Eclipse IDE. The only difference is between configuration steps, and this defines the difference between one Eclipse IDE version and another.

The ADT offers the possibility of using a standalone cross-compiler, debugging tool profilers, emulators, and even development board interaction in a platform-independent manner. Although interaction with hardware is the best option for an embedded developer, in most cases, the real hardware is missing due to various reasons. For these scenarios, it is possible to use a QEMU emulator to simulate the necessary hardware.

The Application Development Toolkit

ADT is one of the components of the Yocto Project and provides a cross-development platform, which is perfect for user-specific application development. For the development process to take place in an orderly manner, some components are required:

- Eclipse IDE Yocto plug-ins
- QEMU emulator for specific hardware simulations
- Cross-toolchain alongside its specific sysroot, which are both architecture-specific and are generated using the metadata and the build system made available by the Yocto Project
- Userspace tools to enhance a developer's experience with the application development process

The Eclipse plug-ins are available when offering full support to the Yocto Project with the Eclipse IDE and maximizing the Yocto experience. The end result is an environment that is customized for the Yocto developer's needs, with a cross-toolchain, deployment on a real hardware, or QEMU emulation features, and also a number of tools that are available for collecting data, tracing, profiling, and performance reviews.

The QEMU emulator is used to simulate various hardware. It can be obtained with these methods:

- Using the ADT installer script, which offers the possibility of installing it
- Cloning a Poky repository and sourcing the environment, access is granted to a QEMU environment
- Downloading a Yocto release and sourcing the environment offers for the same result
- Installing a cross-toolchain and sourcing the environment to make the QEMU environment available

The toolchain contains a cross-debugger, cross-compiler, and cross-linker, which are very well used in the process of application development. The toolchain also comes with a matching sysroot for the target device because it needs access to various headers and libraries necessary to run on the target architecture. The sysroot is generated from the root filesystem and uses the same metadata configuration.

The userspace tools include the tools already mentioned in the previous chapters, such as SystemTap, PowerTop, LatencyTop, perf, OProfile, and LTTng-UST. They are used for getting information about the system and developed application; information, such as power consumption, desktop stutters, counting of events, performance overviews, and diagnosing software, hardware, or functional problems, and even tracing software activities.

Setting up the environment

Before explaining the ADT Project further, its Eclipse IDE plug-ins, other features, and functionalities of the setup would be required. To install the Eclipse IDE, the first step involves the setup of a host system. There are multiple methods to do this:

- **Using an ADT install script:** This is the recommended method to install the ADT, mostly because the installation process is completely automated. Users are in control of the features that they want available.
- **Using the ADT tarball:** This method involves a section of an appropriate tarball with the architecture-specific toolchain and setting it up using a script. The tarball can be both downloaded and manually built using Bitbake. This method also has limitations due to the fact that not all of its features are available after installation, apart from the cross-toolchain and QEMU emulator.
- **Using a toolchain from the build directory:** This method takes advantage of the fact that a build directory is already available, so the setup of the cross-toolchain is very easy. Also, in this case, it faces the same limitation as the one mentioned in the preceding point.

The ADT install script is the preferred method to install the ADT. Of course, before moving on to the installation step, the necessary dependencies need to be available to make sure that the ADT install script runs smoothly.

These packages were already mentioned in the previous chapters, but they will once again, be explained here to make things easy for you. I advise you to go back to these chapters and refer to the information once again as a memory exercise. To refer to packages that might be of interest to you, take a look at the ADT Installer packages, such as `autoconf` `automake` `libtool` `libglib2.0-dev`, Eclipse Plug-ins, and graphical support offered by the `libsdl1.2-dev` `xterm` packages.

After the host system is prepared with all the required dependencies, the ADT tarball can be downloaded from <http://downloads.yoctoproject.org/releases/yocto/yocto-1.7/adt-installer/>. At this location, the `adt_installer.tar.bz2` archive is available. It needs to be downloaded and its content extracted.

This tarball can also be generated using the Bitbake build system inside a build directory, and the result will be available inside the `tmp/deploy/sdk/adt_installer.tar.bz2` location. To generate it, the next command needs to be given into the build directory, which is `bitbake adt-installer`. The build directory also needs to be properly configured for the target device.

The archive is unpacked using the `tar -xjf adt_installer.tar.bz2` command. It can be extracted in any directory, and after unpacking the `adt-installer` directory, it is created and contains the ADT installer script called `adt_installer`. It also has a configuration file called `adt_installer.conf`, which is used to define the configurations before running the script. The configuration file defines information, such as the filesystem, kernel, QEMU support, and so on.

These are the variables that the configuration file contains:

- `YOCTOADT_REPO`: This defines the packages and root filesystem on which the installation is dependent. Its reference value is defined at <http://adtrepo.yoctoproject.org/>. Here, the directory structure is defined and its structure is the same between releases.
- `YOCTOADT_TARGETS`: This defines the target architecture for which the cross development environment is set up. There are default values defined that can be associated with this variable, such as `arm`, `ppc`, `mips`, `x86`, and `x86_64`. Also, multiple values can be associated with it and the separation between them being is done using the space separator.
- `YOCTOADT_QEMU`: This variable defines the use of the QEMU emulator. If it is set to `Y`, the emulator will be available after installation; otherwise the value is set to `N`, and hence, the emulator won't be available.
- `YOCTOADT_NFS_UTIL`: This defines if the NFS user-mode that will be installed. The available values are, as defined previously, `Y` and `N`. For the use of the Eclipse IDE plug-ins, it is necessary to define the `Y` value for both `YOCTOADT_QEMU` and `YOCTOADT_NFS_UTIL`.
- `YOCTOADT_ROOTFS_<arch>`: This specifies which architecture root filesystem to use from the repository that is defined in the first mentioned `YOCTOADT_REPO` variable. For the `arch` variable, the default values are the ones already mentioned in the `YOCTOADT_TARGETS` variable. This variable's valid values are represented by the image files available, such as `minimal`, `sato`, `minimal-dev`, `sato-sdk`, `lsb`, `lsb-sdk`, and so on. For multiple arguments to the variable, the space separator can be used.

- `YCTOADT_TARGET_SYSROOT_IMAGE_<arch>`: This represents the root filesystem from which the `sysroot` of the cross-development toolchain will be generated. The valid values for the '`arch`' variable are the same as the one mentioned previously. Its value is dependent on what was previously defined as values for the `YCTOADT_ROOTFS_<arch>` variable. So, if only one variable is defined as the value for the `YCTOADT_ROOTFS_<arch>` variable, the same value will be available for `YCTOADT_TARGET_SYSROOT_IMAGE_<arch>`. Also, if multiple variables are defined in the `YCTOADT_ROOTFS_<arch>` variable, then one of them needs to define the `YCTOADT_TARGET_SYSROOT_IMAGE_<arch>` variable.
- `YCTOADT_TARGET_MACHINE_<arch>`: This defines the machine for which the image is downloaded, as there could be compilation option differences between machines of the same architecture. The valid values for this variable are can be mentioned as: `qemuarm`, `qemuppc`, `ppc1022ds`, `edgerouter`, `beaglebone`, and so on.
- `YCTOADT_TARGET_SYSROOT_LOC_<arch>`: This defines the location where the target `sysroot` will be available after the installation process.

There are also some variables defined in the configuration files, such as `YCTOADT_BITBAKE` and `YCTOADT_METADATA`, which are defined for future work references. After all the variables are defined according to the needs of the developer, the installation process can start. This is done by running the `adt_installer` script:

```
cd adt-installer  
./adt_installer
```

Here is an example of the `adt_installer.conf` file:

```
# Yocto ADT Installer Configuration File  
#  
# Copyright 2010-2011 by Intel Corp.  
#  
# Permission is hereby granted, free of charge, to any person obtaining a  
copy  
# of this software and associated documentation files (the "Software"),  
to deal  
# in the Software without restriction, including without limitation the  
rights  
# to use, copy, modify, merge, publish, distribute, sublicense, and/or  
sell
```

```
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:

# The above copyright notice and this permission notice shall be included
in

# all copies or substantial portions of the Software.

# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
OR

# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY,

# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
THE

# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM,

# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS
IN

# THE SOFTWARE.

# Your yocto distro repository, this should include IPKG based packages
and root filesystem files where the installation is based on

YOCTOADT_REPO="http://adtrepo.yoctoproject.org//1.7"
YOCTOADT_TARGETS="arm x86"
YOCTOADT_QEMU="Y"
YOCTOADT_NFS_UTIL="Y"

#YOCTOADT_BITBAKE="Y"
#YOCTOADT_METADATA="Y"

YOCTOADT_ROOTFS_arm="minimal sato-sdk"
YOCTOADT_TARGET_SYSROOT_IMAGE_arm="sato-sdk"
YOCTOADT_TARGET_MACHINE_arm="qemuarm"
YOCTOADT_TARGET_SYSROOT_LOC_arm="$HOME/test-yocto/$YOCTOADT_TARGET_
MACHINE_arm"
```

```
#Here's a template for setting up target arch of x86
YOCTOADT_ROOTFS_x86="sato-sdk"
YOCTOADT_TARGET_SYSROOT_IMAGE_x86="sato-sdk"
YOCTOADT_TARGET_MACHINE_x86="qemux86"
YOCTOADT_TARGET_SYSROOT_LOC_x86="$HOME/test-yocto/$YOCTOADT_TARGET_
MACHINE_x86"

#Here's some template of other arches, which you need to change the value
in ""
YOCTOADT_ROOTFS_x86_64="sato-sdk"
YOCTOADT_TARGET_SYSROOT_IMAGE_x86_64="sato-sdk"
YOCTOADT_TARGET_MACHINE_x86_64="qemux86-64"
YOCTOADT_TARGET_SYSROOT_LOC_x86_64="$HOME/test-yocto/$YOCTOADT_TARGET_
MACHINE_x86_64"

YOCTOADT_ROOTFS_ppc="sato-sdk"
YOCTOADT_TARGET_SYSROOT_IMAGE_ppc="sato-sdk"
YOCTOADT_TARGET_MACHINE_ppc="qemuppc"
YOCTOADT_TARGET_SYSROOT_LOC_ppc="$HOME/test-yocto/$YOCTOADT_TARGET_
MACHINE_ppc"

YOCTOADT_ROOTFS_mips="sato-sdk"
YOCTOADT_TARGET_SYSROOT_IMAGE_mips="sato-sdk"
YOCTOADT_TARGET_MACHINE_mips="qemumips"
YOCTOADT_TARGET_SYSROOT_LOC_mips="$HOME/test-yocto/$YOCTOADT_TARGET_
MACHINE_mips"
```

After the installation has started, the user is asked the location of the cross-toolchain. If no alternative is offered, the default path is selected and the cross-toolchain is installed in the `/opt/poky/<release>` directory. The installation process can be visualized both in a silent or interactive way. By using the `I` option, the installation is done in an interactive mode, while the silent mode is enabled using the `S` option.

At the end of the install procedure, the cross-toolchain will be found in its defined location. An environment setup script will be available for later usage, and the image tarball in the `adt-installer` directory, and the `sysroot` directory is defined in the location of the `YOCTOADT_TARGET_SYSROOT_LOC_<arch>` variable.

As shown previously, there is more than one method to prepare the ADT environment. The second method involves only the installation of the toolchain installer—although it offers the possibility of having a prebuilt cross-toolchain, support files and scripts, such as the `rungemu` script to start something similar to a kernel or Linux image in an emulator—which does not offer the same possibilities as the first option. Also, this option has its limitations regarding the `sysroot` directory. Although it's been generated, the `sysroot` directory might still need to be extracted and installed in a separate location. This can happen for various reasons, such as the need to boot a root filesystem over NFS or develop the application using the root filesystem as the target `sysroot`.

The root filesystem can be extracted from an already generated cross-toolchain using the `rungemu-extract-sdk` script, which should be called only after the cross-development environment script was set up using `source` command.

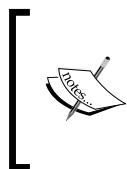
There are two methods to obtain the toolchain installed for this second option. The first method involves the use of the toolchain installer available at <http://downloads.yoctoproject.org/releases/yocto/yocto-1.7/toolchain/>. Open the folder that matches your development host machine. In this folder, multiple install scripts are available. Each one matches a target architecture, so the right one should be selected for the target you have. One such example can be seen from http://downloads.yoctoproject.org/releases/yocto/yocto-1.7/toolchain/x86_64/poky-glibc-x86_64-core-image-sato-armv7a-vfp-neon-toolchain-1.7.sh, which is, in fact, the installer script for the `armv7a` target and the `x86_64` host machine.

If your target machine is not one of the ones that are made available by the Yocto community, or if you prefer an alternative to this method, then building the toolchain installer script is the method for you. In this case, you will require a build directory, and you will be presented with two alternatives, both of them are equally good:

- The first one involves the use of the `bitbake meta-toolchain` command, and the end result is an installer script that requires the installation and set up of the cross-toolchain in a separate location.
- The second alternative involves the use of the `bitbake -c populate_sdk <image-name>` task, which offers the toolchain installer script and the matching `sysroot` for the target. The advantage here is that the binaries are linked with only one and the same `libc`, making the toolchain self-contained. There is, of course, a limitation that each architecture can create only one specific build. However, target-specific options are passed through the `gcc` options. Using variables, such as `CC` or `LD`, makes the process easier to maintain and also saves some space in the build directory.

After the installer is downloaded, make sure that the install script has set the execution correctly, and start the installation with the `./poky-glibc-x86_64-core-image-sato-armv7a-vfp-neon-toolchain-1.7.sh` command.

Some of the information you require includes the place where the installation should be made, the default location being the `/opt/poky/1.7` directory. To avoid this, the script can be called with the `-d <install-location>` argument and the installation can be made in the `<install-location>` location, as mentioned.



Make sure that the `MACHINE` variable is set accordingly in the `local.conf` file. Also, if the build is done for a different host machine, then `SDKMACHINE` should also be set. More than one `MACHINE` cross-toolchain can be generated in the same build directory, but these variables need to be properly configured.

After the installation process is finished, the cross-toolchain will be available in the selected location, and the environment script will also be available for sourcing when needed.

The third option involves the use of the build directory and the execution of the `bitbake meta-ide-support` command. Inside the build directory, the proper environment needs to be set using one of the two available build environment setup scripts, which include the `oe-init-build-env` script or `oe-init-build-env-memres`. The local configuration from the `local.conf` file also needs to be set accordingly for the target architecture. After these steps are fulfilled by the developer, the `bitbake meta-ide-support` command could be used to start the generation of the cross-toolchain. At the end of the process, an environment setup script will be available inside the `<build-dir-path>/tmp` directory, but in this case, the toolchain is tightly linked into the build directory in which it was built.

With the environment set up, writing of an application can start, but the developer would still need to complete some steps before finishing the activity, such as testing the application on the real root filesystem, debugging, and many others. For the kernel module and driver implementation, the kernel source code will be required, so the activity is just starting.

Eclipse IDE

The plug-ins available for Eclipse from the Yocto Project include the functionalities for the ADT Project and toolchain. They allow developers to use a cross-compiler, debugger, and all the available tools generated with the Yocto Project, Poky, and additional meta layers. Not only can these components be used within the Eclipse IDE, but they also offer a familiar environment for application development.

The Eclipse IDE is an alternative for developers who are not interested in interacting with editors, such as `vim`, although, in my opinion, `vim` can be used for all kinds of projects. Even if their dimensions or complexities are not a problem, the overhead for using `vim` might not suit all tastes. The Eclipse IDE is the best alternative available for all developers. It has a lot of useful features and functionalities that can make your life a little easier and it is pretty easy to grasp.

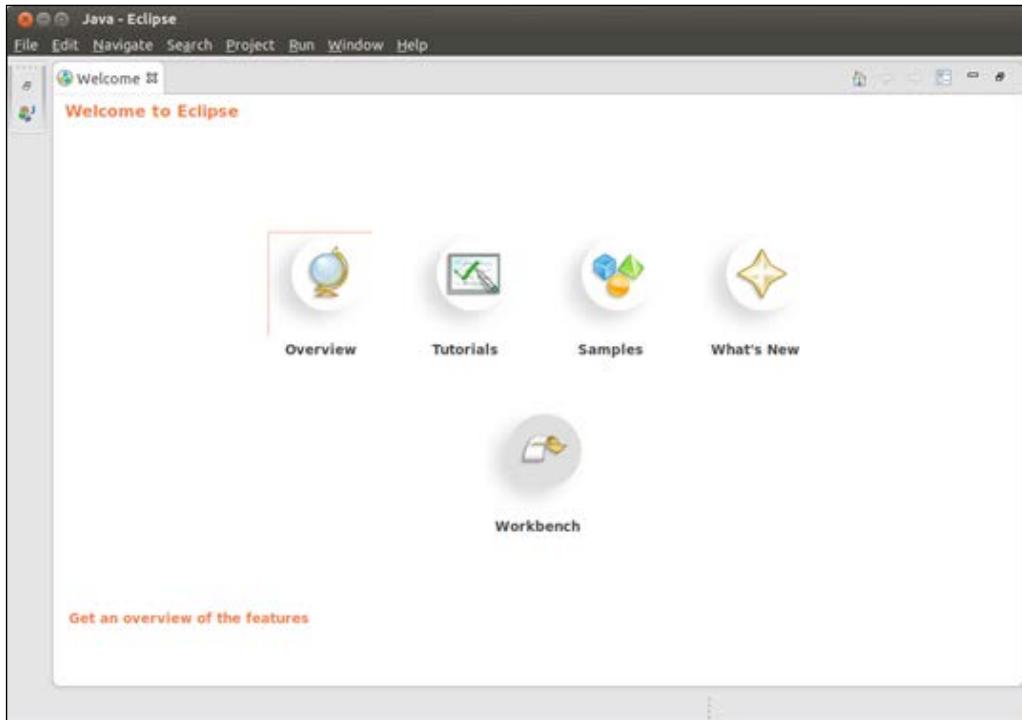
The Yocto Project offers support for two versions of Eclipse, Kepler and Juno. The Kepler version is the one recommended with the latest Poky release. I also recommend the Kepler 4.3.2 version of Eclipse, the one downloaded from the official download site of Eclipse, <http://www.eclipse.org/downloads>.

From this site, the Eclipse Standard 4.3.2 version containing the **Java Development Tools (JDT)**, the Eclipse Platform, and the Development Environment Plug-ins for the host machine should be downloaded. After the download is finished, the received archive content should be extracted using the tar command:

```
tar xzf eclipse-standard-kepler-SR2-linux-gtk-x86_64.tar.gzls
```

The next step is represented by the configuration. With the content extracted, the Eclipse IDE needs to be configured before installing the Yocto Project-specific plug-ins. The configuration starts with initializing the Eclipse IDE:

The Eclipse IDE is started after executing the `./eclipse` executable and setting the Workspace location. This is how the starting windows looks:

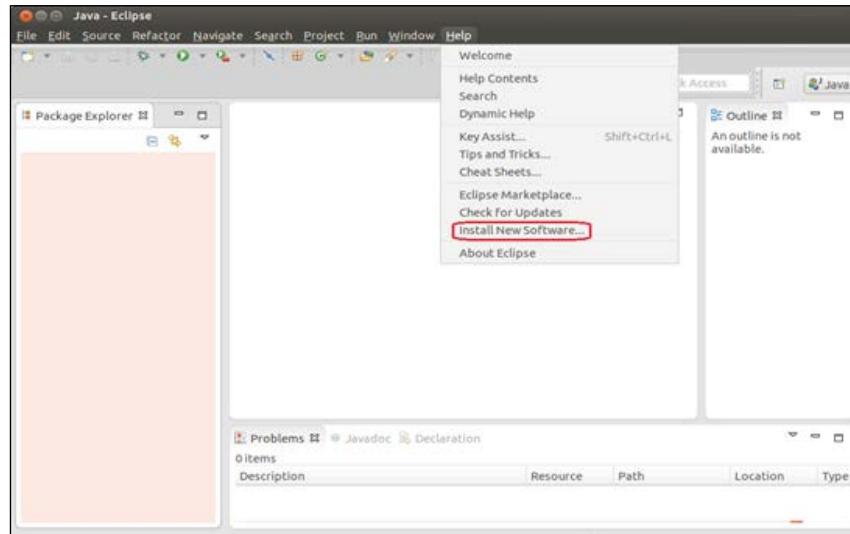


Eclipse window

To initialize the Eclipse IDE perform the following steps:

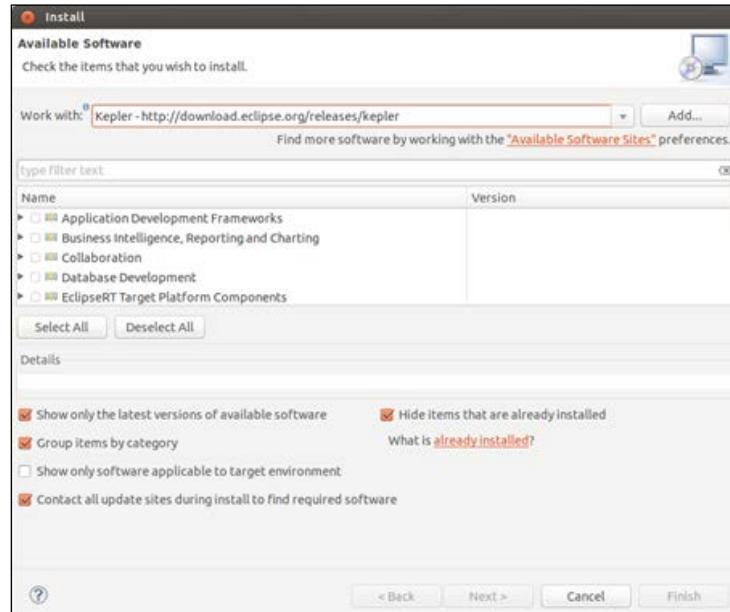
1. Select **Workbench**, and you will be moved into the empty workbench where the projects source code will be written.

2. Now, navigate through the **Help** menu and select **Install New Software**.



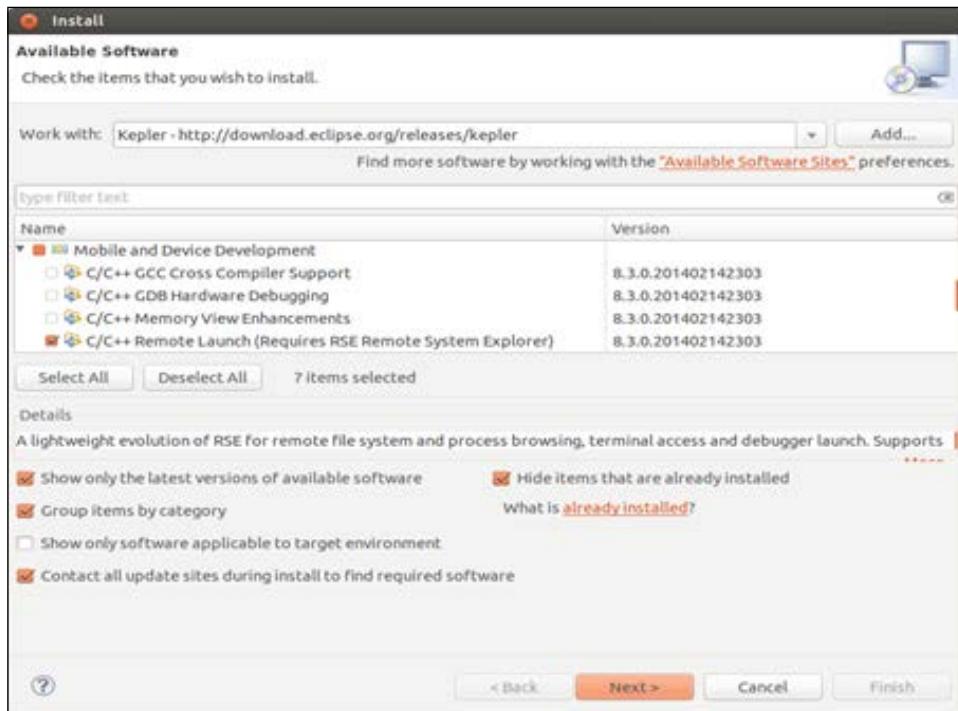
Help menu

3. A new window will open, and in the **Work with:** drop-down menu, select **Kepler - <http://download.eclipse.org/releases/kepler>**, as shown in the following screenshot:



Install window

4. Expand the **Linux Tools** section and select **LTTng - Linux Tracing Toolkit** box, as shown in the following screenshot:

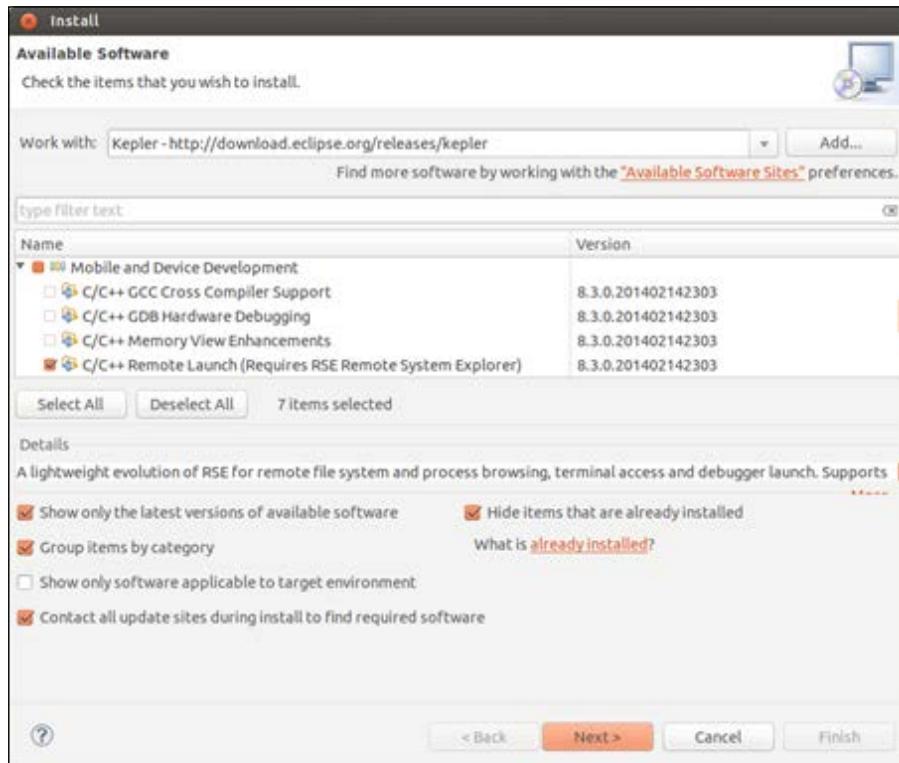


Install – LTTng – Linux Tracing Toolkit box

5. Expand the **Moble and Device Development** section and select the following:
 - **C/C++ Remote Launch (Requires RSE Remote System Explorer)**
 - **Remote System Explorer End-user Runtime**
 - **Remote System Explorer User Actions**
 - **Target Management Terminal**

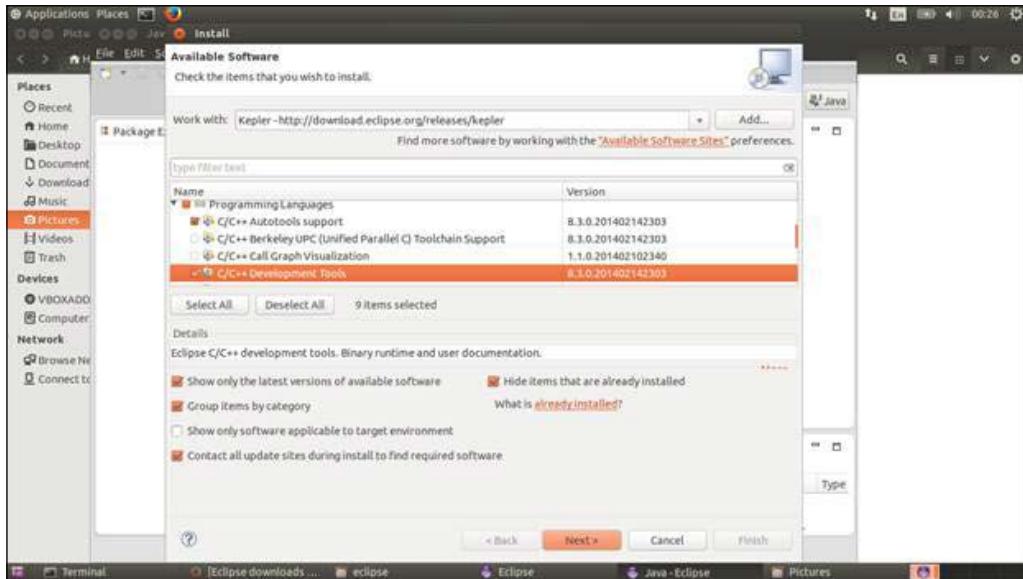
ADT Eclipse Plug-ins

- **TCF Remote System Explorer add-in**
- **TCF Target Explorer**



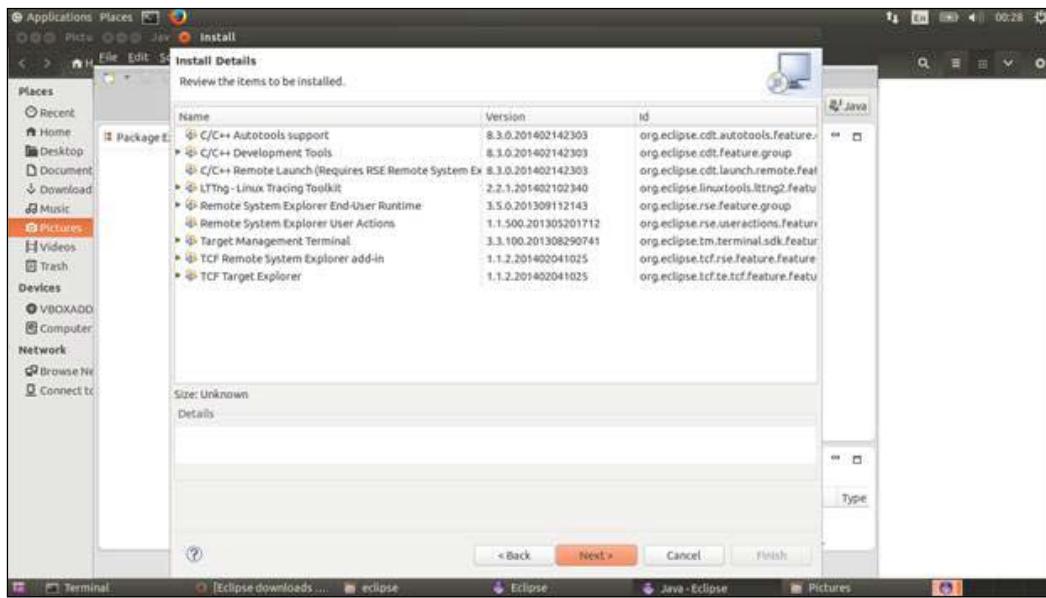
6. Expand the **Programming Languages** section and select the following:
- **C/C++ Autotools Support**
 - **C/C++ Development Tools**

This is shown in the following screenshot:



Available software list window

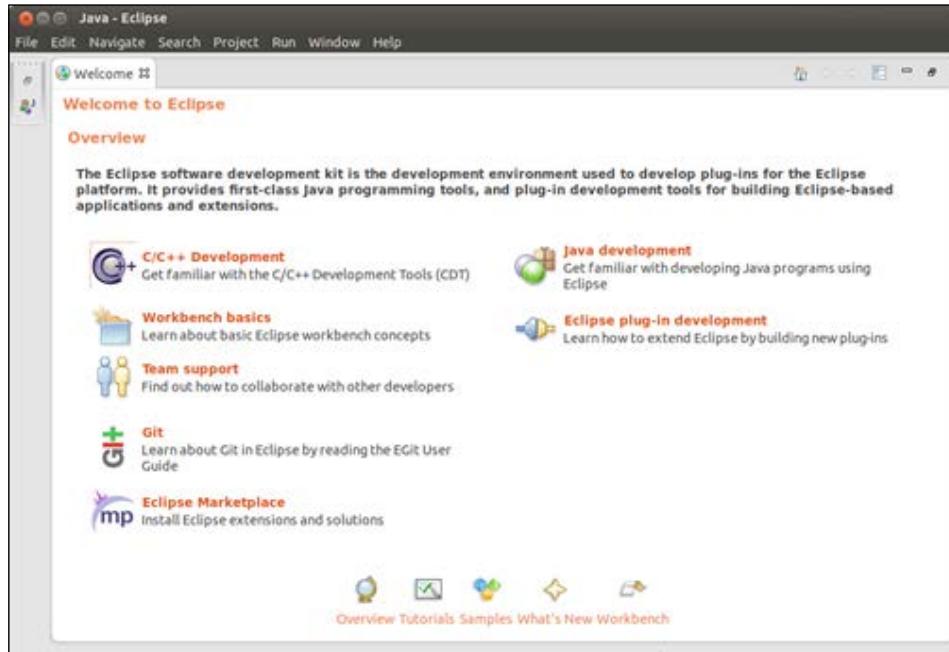
7. Finish the installation after taking a quick look at the **Install Details** menu and enabling the license agreement:



Install details window

ADT Eclipse Plug-ins

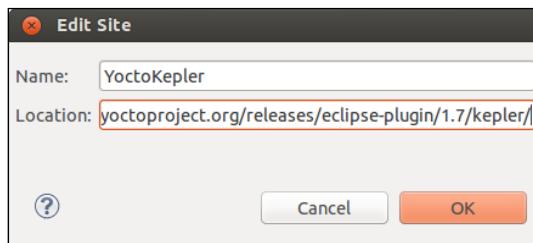
After these steps, the Yocto Project Eclipse plug-ins can be installed into the IDE, but not before restarting the Eclipse IDE to make sure that the preceding changes take effect. The result after the configuration phase is visible here:



Eclipse – Configuring phase results

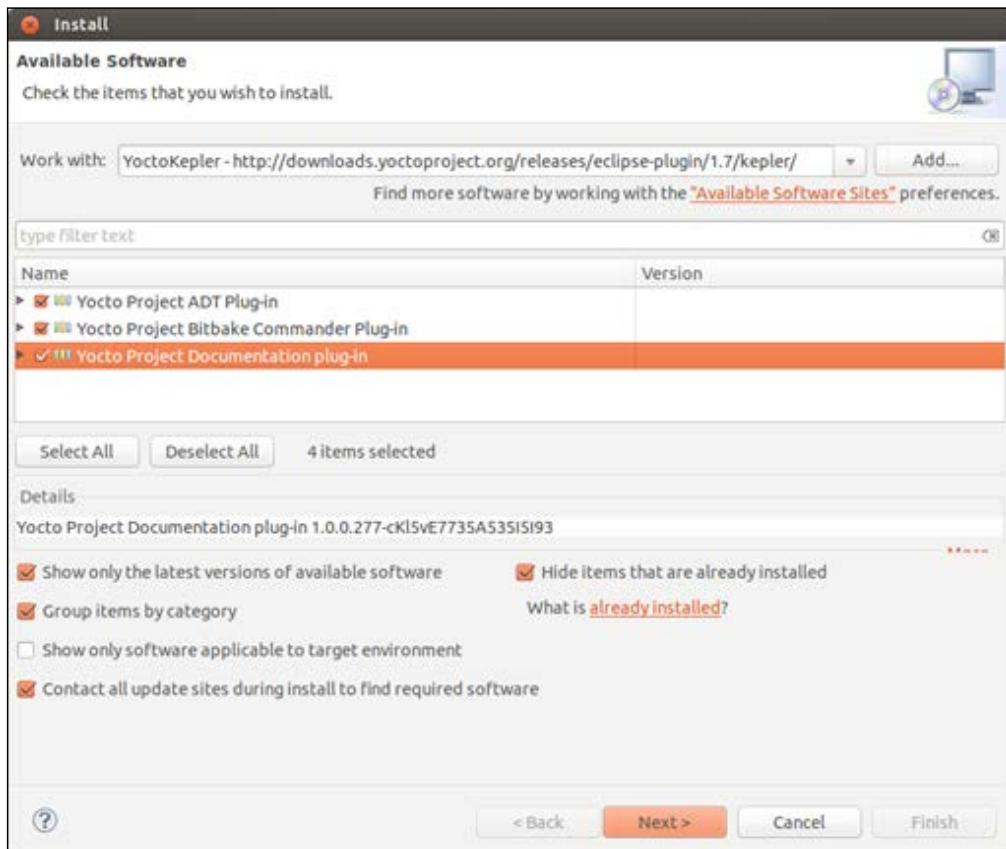
To install the Eclipse plug-ins for the Yocto Project, these steps are required:

1. Start the Eclipse IDE as mentioned previously.
2. As shown in the previous configuration, select the **Install New Software** option from the **Help** menu.
3. Click on the **Add** button and insert `downloads.yoctoproject.org/releases/eclipse-plugin/1.7/kepler/` in the URL section. Give a proper name to the new **Work with:** site as indicated here:



Edit site window

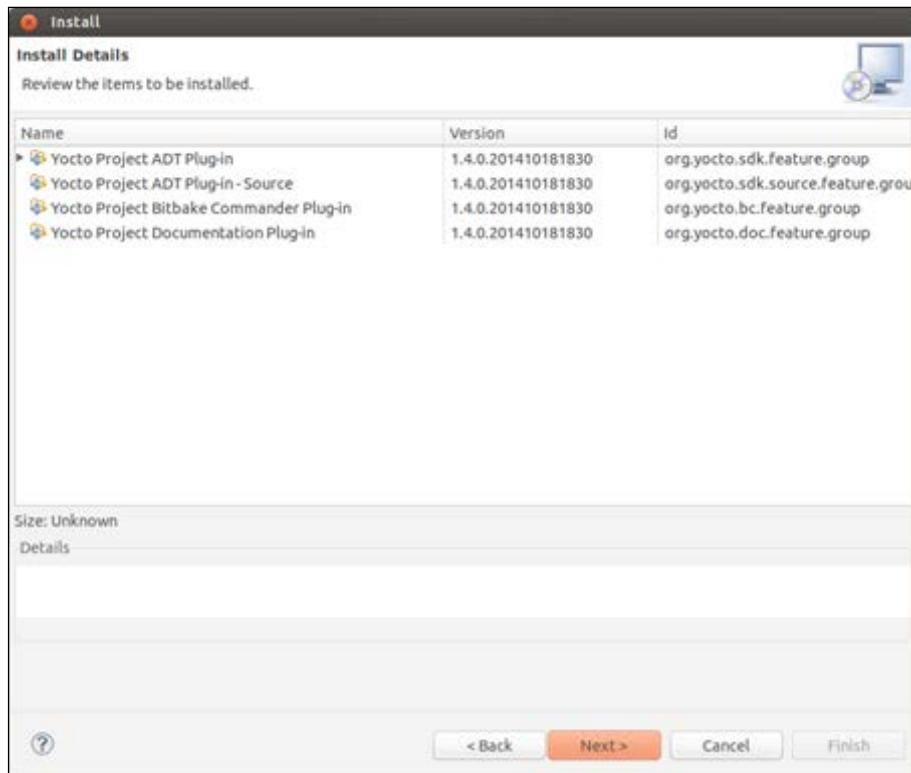
- After the **OK** button is pressed, and the **Work with** site is updated, new boxes appear. Select all of them, as shown in this image, and click on the **Next** button:



Install details window

ADT Eclipse Plug-ins

5. One final pick at the installed components and the installation is approaching its end.



Install details window

6. If this warning message appears, press **OK** and move further. It only lets you know that the installed packages have unsigned content.

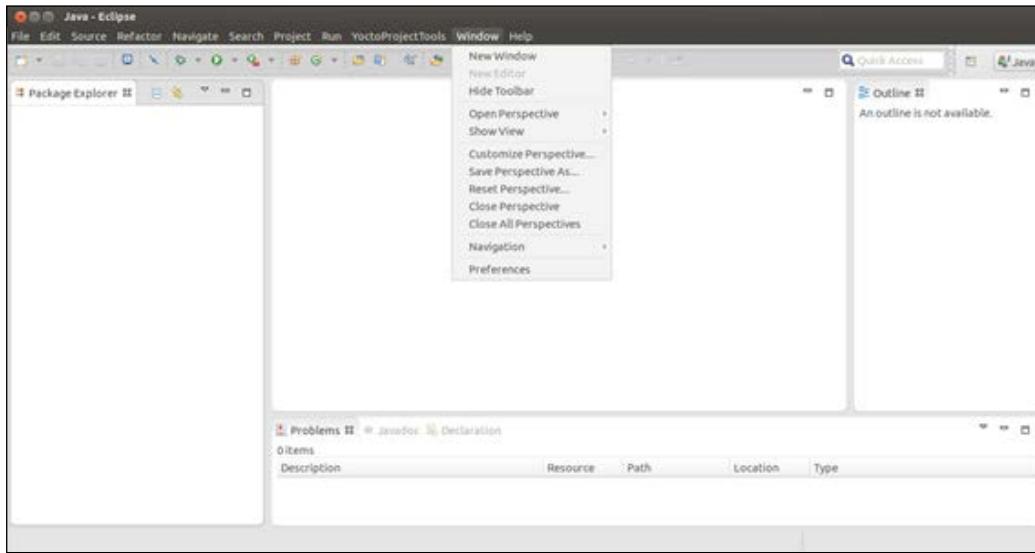


Security warning window

The installation finishes only after the Eclipse IDE is restarted for the changes to take effect.

After the installation, the Yocto plug-ins are available and ready to be configured. The configuration process involves the setup of the target-specific option and cross-compiler. For each specific target, the preceding configurations steps need to be performed accordingly.

The configuration process is done by selecting the **Preferences** option from the **Window** menu. A new window will open, and from there, the **Yocto Project ADT** option should be selected. More details are available, as shown in the following screenshot:



Eclipse IDE – Preferences

The next thing to do involves the configuration of the available options of the cross-compiler. The first option refers to the toolchain type, and there are two options available, **Standalone prebuilt toolchain** and **Build system derived toolchain**, which is the default selected option. The former refers to a toolchain specific for an architecture that already has an existing kernel and root filesystem, so the developed application will be made available in the image manually. However, this step is not a requirement since all the components are separated. The latter option refers to a toolchain built inside a Yocto Project build directory.

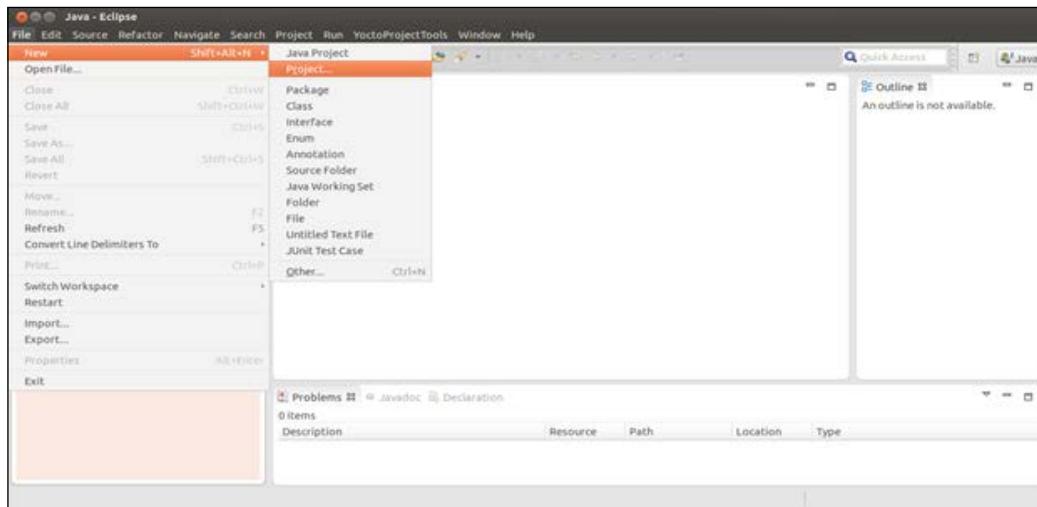
The next elements that need to be configured are the toolchain location, `sysroot` location, and the target architecture. The **Toolchain Root Location** is used to define the toolchain install location. For an installation done with the `adt_installer` script, for example, the toolchain will be available in the `/opt/poky/<release>` directory. The second argument, **Sysroot Location**, represents the location of the target device root filesystem. It can be found in the `/opt/poky/<release>` directory, as seen in the preceding example, or even inside the build directory if other method to generate it were used. The third and last option from this section is represented by the **Target Architecture** and it indicates the type of hardware used or emulated. As it can be seen on the window, it is a pull-down menu where the required option is selected, and a user will find all the supported architectures listed. In a situation where the necessary architecture is not available inside the pull-down menu, the corresponding image for the architecture will need to be built.

The last remaining section is represented by the target specific option. This refers to the possibility of emulating an architecture using QEMU or running the image on the externally available hardware. For external hardware, use the **External HW** option that needs to be selected for the work to be finished, but for the QEMU emulation, there are still things to do besides selecting the **QEMU** option. In this scenario, the user will also need to indicate the **Kernel** and **Custom Option**. For the kernel selection, the process is simple. It is available in the prebuilt image location in case the **Standalone pre-built toolchain** option was selected or in the `tmp/deploy/images/<machine-name>` directory if the **Build system derived toolchain** option was selected. For the second option, the **Custom Option** argument, the process for adding it will not be as simple as the preceding options.

The **Custom Option** field needs to be filled with various options, such as `kvm`, `nographic`, `publicvnc`, or `serial`, which indicate major options for the emulated architecture or their parameters. These are kept inside angled brackets, and include parameters, such as the memory used (`-m 256`), networking support (`-net`), and full screen support (`-full-screen`). More information regarding the available options and parameters can be found using the `man qemu` command. All of the preceding configurations can be overwritten using the **Change Yocto Project Settings** option from the **Project** menu after a project is defined.

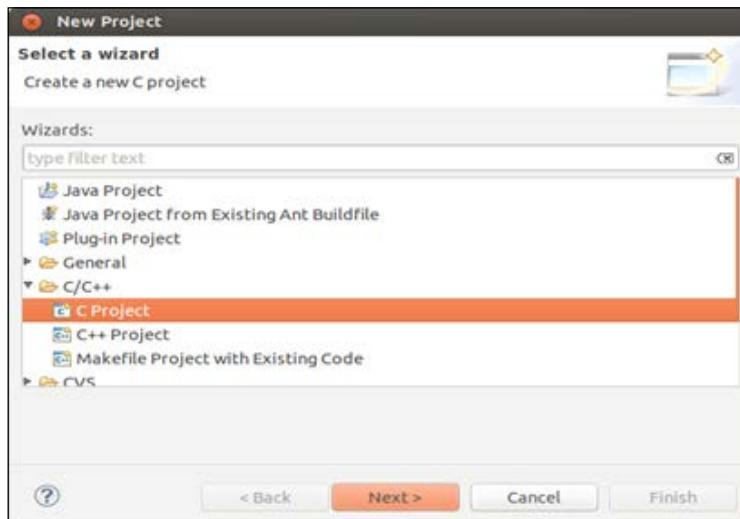
To define a project, these steps need to be taken:

1. Select the **Project...** option from the **File | New** menu option, as shown here:



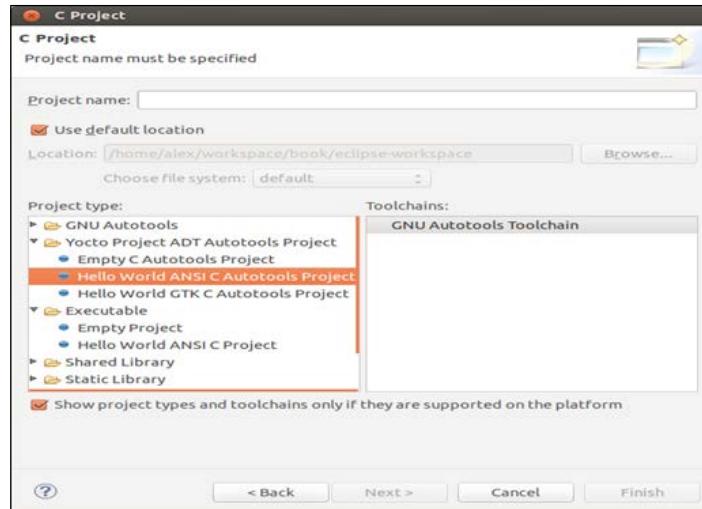
Eclipse IDE – Project

2. Select **C project** from the **C/C++** option. This will open a **C Project** window:



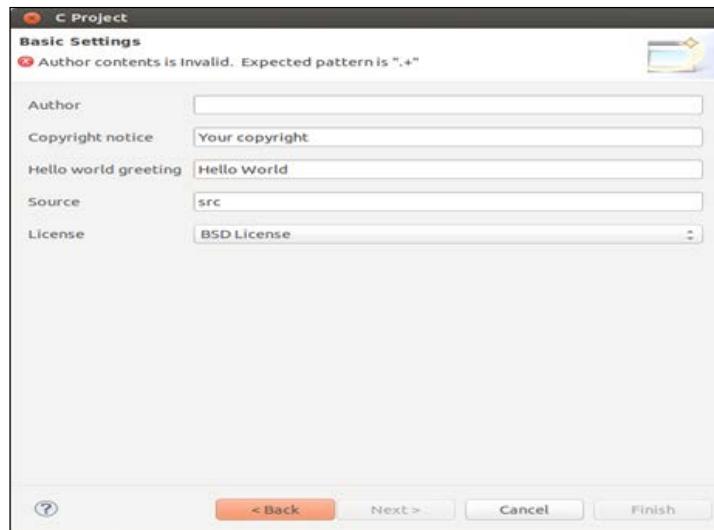
Eclipse IDE – New project window

3. In the **C Project** window, there are multiple options available. Let's select **Yocto Project ADT Autotools Project**, and from there, the **Hello World ANSI C Autotools Project** option. Add a name for the new project, and we are ready to move to the next steps:



C project window

4. In the **C Project** window we you be prompted to add information regarding the **Author**, **Copyright notice**, **Hello world greetings**, **Source**, and **License** fields accordingly:



C project – basic settings window

5. After all the information is added, the **Finish** button can be clicked on. The user will be prompted in the new **C/C++** perspective that is specific for the opened project, with the newly created project appearing on the left-hand side of the menu.
6. After the project is created and the source code is written, to build the project, select the **Build Project** option from the **Project...** menu.

QEMU emulator

QEMU is used in the Yocto Project as a virtualization machine and emulator for various target architectures. It is very useful to run and test various Yocto generated applications and images, apart from fulfilling other purposes. Its primary use outside of the Yocto world is its selling point for the Yocto Project too, making it the default tool to be used to emulate hardware.



More information about the use case of QEMU can be found by accessing <http://www.yoctoproject.org/docs/1.7/adt-manual/adt-manual.html#the-qemu-emulator>.



Interaction with the QEMU emulation is done within Eclipse, as shown previously. For this to happen, the proper configuration would be required, as instructed in the preceding section. Starting the QEMU emulation here is done using the **External Tools** option from the **Run** menu. A new window will be opened for the emulator, and after the corresponding login information is passed to the prompt, the shell will be available for user interaction. An application can be deployed and debugged on the emulator also.



More information regarding QEMU interaction is available at <http://www.yoctoproject.org/docs/1.7/dev-manual/dev-manual.html#dev-manual-qemu>.



Debugging

Debugging an application can also be done using the QEMU emulator or the actual target hardware, if it exists. When the project was configured, a run/debug Eclipse configuration was generated as a **C/C+ Remote Application** instance, and it can be found on the basis of its name, which is according to the `<project-name>_gdb-<suffix>` syntax. For example, `TestProject_gdb_armv5te-poky-linux-gnueabi` could be an example of this.

To connect to the Eclipse GDB interface and start the remote target debugging process, the user is required to perform a few steps:

1. Select **C/C++ Remote application** from the **Run | Debug configuration** menu and choose the run/debug configuration from the **C/C++ Remote Application** available in the left panel.
2. Select the suitable connection from the drop-down list.
3. Select the binary application to deploy. If multiple executables are available in your project, by pushing the **Search Project** button, Eclipse will parse the project and provide a list with all the available binaries.
4. Enter the absolute path in which the application will be deployed by setting the **Remote Absolute File Path for C/C++ Application:** field accordingly.
5. Selecting the debugger option is available in the **Debugger** tab. To debug shared libraries, a few extra steps are necessary:
 - Select the **Add | Path Mapping** option from the **Source** tab to make sure a path mapping is available for the debug configuration.
 - Select **Load shared libraries symbols automatically** from the **Debug/Shared Library** tab and indicate the path of the shared libraries accordingly. This path is highly dependent on the architecture of the processor, so be very careful which library file you indicate. Usually, for the 32-bit architecture, the `lib` directory is selected, and for the 64-bit architecture, the `lib64` directory is chosen.
 - On the **Arguments** tab, there is a possibility of passing various arguments to the application binary during the time of execution.
6. Once all the debug configurations are finished, click on the **Apply** and **Debug** buttons. A new GDB session will be launched and **Debug perspective** will open. When the debugger is being initialized, Eclipse will open three consoles:
 - A GDB console named after the GDB binary described previously, used for command-line interaction
 - A remote shell used to run an application display results
 - A local machine console that is named after the binary path, which in most of cases, is not used. It remains as an artefact.
7. After the setup of the debug configuration, the application can be rebuilt and executed again using the available **Debug** icon in the toolbar. If, in fact, you want only to run and deploy the application, the **Run** icon can be used.

Profiling and tracing

Inside the **Yocto Tools** menu, you can see the supported tools that are used for the tracing and profiling of developed applications. These tools are used for enhancing various properties of the application and, in general, the development process and experience. The tools that will be presented are LTTng, Perf, LatencyTop, PerfTop, SystemTap, and KGDB.

The first one we'll take a look at is the LTTng Eclipse Plug-in, which offers the possibility of tracing a target session and analyzing the results. To start working with the tool, a quick configuration is necessary first, as follows:

1. Start the tracing perspective by selecting **Open Perspective** from the **Window** menu.
2. Create a new tracing project by selecting **Project** from the **File | New** menu.
3. Select **Control View** from the **Window | Show view | Other... | Lttn** menu. This will enable you to access all these desired operations:
 - Creating a new connection
 - Creating a session
 - Starting/stopping tracing
 - Enabling events

Next, we'll introduce the user space performance analyzing tool called **Perf**. It offers statistical profiling of the application code and a simple CPU for multiple threads and kernel. To do this, it uses a number of performance counters, dynamic probes, or trace points. To use the Eclipse Plug-in, a remote connection to the target is required. It can be done by the Perf wizard or by using the **Remote System Explorer | Connection** option from the **File | New | Other** menu. After the remote connection is set up, interaction with the tool is the same as in the case of the command line support available for the tool.

LatencyTop is an application that is used to identify the latencies available within the kernel and also their root cause. This tool is not available for ARM kernels that have **Symmetric multiprocessing (SMP)** support enabled due to the limitation of the ARM kernels. This application also requires a remote connection. After the remote connection is set up, the interaction is the same as in the case of the command line support available for the tool. This application is run from the Eclipse Plug-in using sudo.

PowerTop is used to measure the consumption of electrical power. It analyzes the applications, kernel options, and device drivers that run on a Linux system and estimates their power consumption. It is very useful to identify components that use the most amount of power. This application requires a remote connection. After the remote connection is set up, the interaction with the application is the same as for the command line available support for the tool. This application is run from the Eclipse Plug-in using the -d option to display the output in the Eclipse window.

SystemTap is a tool that enables the use of scripts to get results from a running Linux. SystemTap provides free software (GPL) infrastructure to simplify the gathering of information about the running Linux system via the tracing of all kernel calls. It's very similar to dtrace from Solaris, but it is still not suited for production systems, unlike dtrace. It uses a language similar to awk and its scripts have the .stp extension. The monitored data can be extracted and various filters and complex processing can be done on them. The Eclipse Plug-in uses the crosstap script to translate the .stp scripts to a C language to create a Makefile, run a C compiler to create a kernel module for the target architecture that is inserted into the target kernel, and later, collect the tracing data from the kernel. To start the SystemTap plug-in in Eclipse, there are a number of steps to be followed:

1. Select the **systemtap** option from the **Yocto Project Tools** menu.
2. In the opened windows, the crosstap argument needs to be passed:
 - Set the **Metadata Location** variable to the corresponding poky directory
 - Set **Remote User ID** by entering the root (the default option) because it has ssh access to the target-any other user that has the same privileges is also a good choice
 - Set in the **Remote Host** variable to the corresponding IP address for the target
 - Use the **Systemtap Scripts** variable for the full path to the .stp scripts
 - Set additional cross options using the **Systemtap Args** field

The output of the .stp script should be available in the console view from Eclipse.

The last tool we'll take a look at is **KGDB**. This tool is used specifically for the debugging of Linux kernel, and is useful only if development on the Linux kernel source code is done inside the Eclipse IDE. To use this tool, a number of necessary configuration setups are required:

- Disable the C/C++ indexing:
 - Select the **C/C++ Indexer** option from the **Window | Preferences** menu
 - Unselect the **Enable indexer** checkbox
- Create a project where the kernel source code can be imported:
 - Select the **C/C++ | C Project** option from the **File | New** menu
 - Select the **Makefile project | Empty project** option and give a proper name to the project
 - Unselect the **Use default location** option
 - Click on the **Browse** button and identify the kernel source code local git repository location
 - Press the **Finish** button and the project should be created

After the prerequisites are fulfilled, the actual configuration can start:

- Select the **Debug Configuration** option from the **Run** menu.
- Double-click on the **GDB Hardware Debugging** option to create a default configuration named **<project name> Default**.
- From the **Main** tab, browse to the location of the `vmlinux` built image, select the **Disable auto build** radio button, as well as the **GDB (DFS) Hardware Debugging Launcher** option.
- For the **C/C++ Application** option available in the **Debugger** tab, browse for the location of the GDB binary available inside the toolchain (if ADT installer script is available, its default location should be `/opt/poky/1.7/sysroots/x86_64-pokysdk-linux/usr/bin/arm-poky-linux-gnueabi/arm-poky-linux-gnueabi-gdb`). Select **Generic serial option** from the **JTAG Device** menu. The **Use remote target** option is a requirement.
- From the **Startup** tab, select the **Load symbols** option. Make sure that the **Use Project binary** option indicates the correct `vmlinux` image and that the **Load image** option is not selected.

- Press the **Apply** button to make sure the previous configuration is enabled.
- Prepare the target for the serial communication debugging:
 - Set the `echo ttys0,115200 | /sys/module/kgdboc/parameters/kgdboc` option to make sure the appropriate device is used for debugging
 - Start KGDB on the `echo g | /proc/sysrq-trigger` target
 - Close the terminal with the target but keep the serial connectivity
- Select the **Debug Configuration** option from the **Run** menu
- Select the previously created configuration and click on the **Debug** button

After the **Debug** button is pressed, the debug session should start and the target will be halted in the `kgdb_breakpoint()` function. From there, all the commands specific to GDB are available and ready to be used.

The Yocto Project bitbake commander

The bitbake commander offers the possibility of editing recipes and creating a metadata project in a manner similar to the one available in the command line. The difference between the two is that the Eclipse IDE is used to do the metadata interaction.

To make sure that a user is able to do these sort of actions, a number of steps are required:

- Select the **Project** option from the **File | New** menu
- Select the **Yocto Project BitBake Commander** wizard from the opened window
- Select the **New Yocto Project** option and a new window will be opened to define properties of the new project
- Using **Project Location**, identify the parent of the `poky` directory
- Use the **Project Name** option to define the project name. Its default value is `poky`
- For the **Remote service provider** variable, select the **Local** choice and make use of the same choice for the **Connection name** drop-down list
- Make sure that the **Clone** checkbox is not selected for an installed `poky` source directory

By using the Eclipse IDE, its features are available to be used. One of the most useful features is the quick search option that could prove to be very useful for some developers. Other benefits include the possibility of creating recipes using templates, editing them with syntax highlighting, auto completion, error reports on the fly, and many more.



The use of bitbake commander is restricted to local connections only. The remote connection causes the IDE to freeze due to a bug available upstream.



Summary

In this chapter, you were presented with information about the functionalities of the ADE offered by the Yocto Project, and the numerous Eclipse Plug-ins available for application development not only as an alternative, but also as a solution for developers who are connected to their IDEs. Although the chapter started with an introduction to the application development options for the command-line enthusiast, it shortly became more about IDE interaction than anything else. This happened because alternative solutions need to be available so that developers could choose what fits their needs best.

In the next chapter, a number of Yocto Project components will be presented. This time, they are not related to application development, but involve metadata interaction, quality assurance, and continuous integration services. I will try to present yet another face of the Yocto Project that I believe will help readers get a better picture of the Yocto Project, and eventually, interact with and contribute to the components that suit them and their needs best.

8

Hob, Toaster, and Autobuilder

In this chapter, you will be introduced to new tools and components used in the Yocto community. As the title suggests, this chapter is dedicated to another category of tools. I will start with **Hob** as a graphical interface, which is slowly dying, and in time, will be replaced by a new web interface called **Toaster**. A new point of discussion will also be introduced in this chapter. Here, I am referring to the QA and testing component that is, in most cases, absent or lacking from most of the projects. Yocto takes this problem very seriously and offers a solution for it. This solution will be presented in the last section of the chapter.

You will also be offered a more detailed presentation to components, such as Hob, Toaster, and Autobuilder. Each of these components will be assessed separately and their benefits and use cases are looked at in detail. For the first two components, (that is, Hob and Toaster) information regarding the build process is offered alongside the various setup scenarios. Hob is similar to BitBake and is tightly integrated with Poky and the Build Directory. Toaster, on the other hand, is a looser alternative that offers multiple configuration alternatives and setups, and a performance section that can be very useful for any developer interested in improving the build system's overall performance. The chapter ends with section on Autobuilder. This project is the cornerstone of the Yocto project that is dedicated to making embedded development and open source more user-friendly, in general, but also offers more secure and error-free projects. I hope that you enjoy this chapter; let's proceed to the first section.

Hob

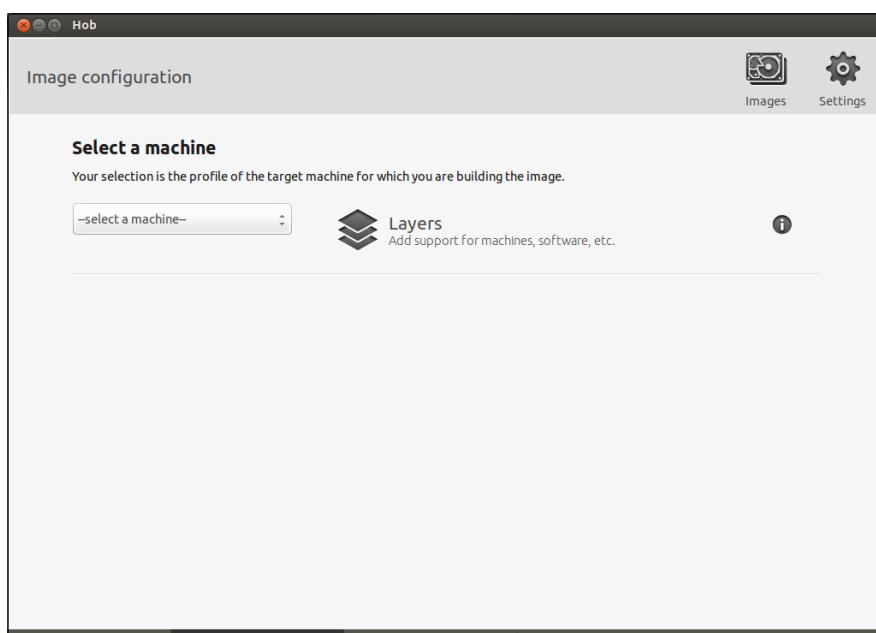
The Hob project represents a GUI alternative to the BitBake build system. Its purpose is to execute the most common tasks in an easier and faster manner, but it does not make command-line interactions go away. This is because most parts of recipes and configurations still need to be done manually. In the previous chapter, the BitBake Commander extension was introduced as an alternative solution for the editing of recipes, but in this project, it has its limitations.

Hob's primary purpose is to allow interaction with the build system made easier for users. Of course, there are users who do not prefer the graphical user interface alternatives to command-line options, and I kind of agree with them, but this is another discussion altogether. Hob can be an option for them also; it is an alternative not only for people who prefer having an interface in front of them, but also for those who are attached to their command-line interaction.

Hob may not be able to a lot of tasks apart from most common ones, such as building an image, modifying its existing recipes, running an image through a QEMU emulator, or even deploying it on a USB device for some live-booting operations on a target device. Having all these functionalities is not much, but is a lot of fun. Your experience with the tools in Yocto Project do not matter here. The previously mentioned tasks can be done very easily and in an intuitive manner, and this is the most interesting thing about Hob. It offers its users what they need in a very easy fashion. People who interact with it can learn from the lessons it has to offer, whether they're graphic interface enthusiasts or command-line savvy.

In this chapter, I will show you how to use the Hob project to build a Linux operating system image. To demonstrate this, I will use the Atmel SAMA5D3 Xplained machine, which is what I also used for other demonstrations in previous chapters.

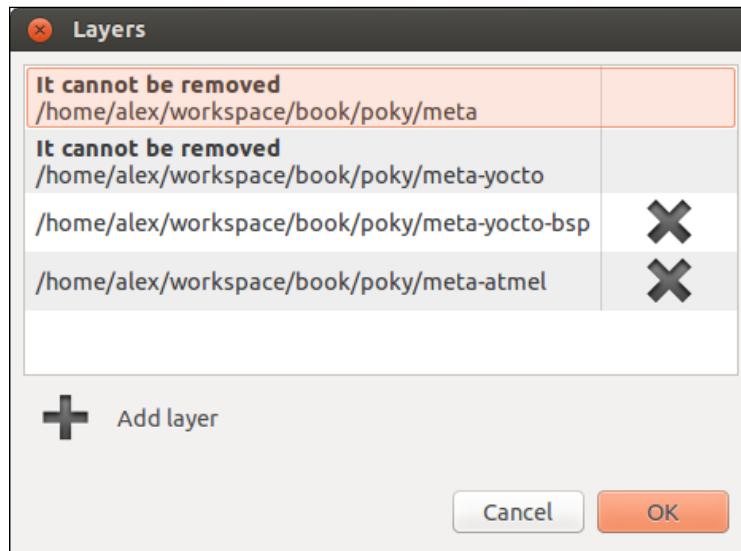
First of all, let's see what Hob looks like when you start it for the first time. The result is shown in the following screenshot:



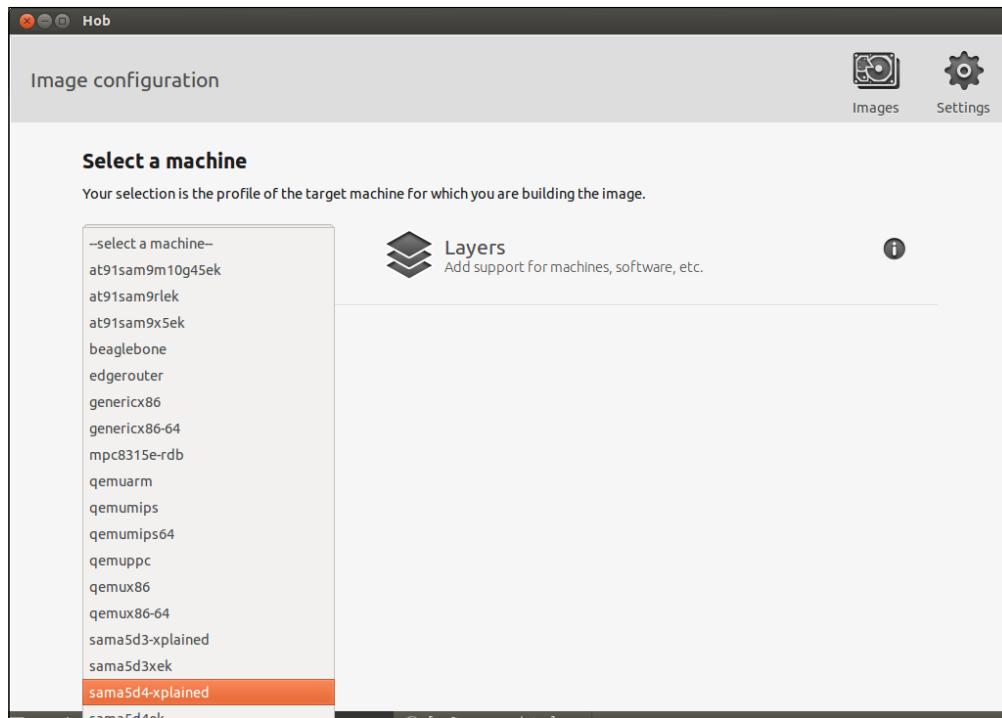
To retrieve the graphical interface, the user needs perform the given steps required for the BitBake command-line interaction. Firstly, it needs to create a build directory and from this build directory, the user needs to start the Hob graphical interface, using the Hob commands, given as follows:

```
source poky/oe-init-build-env ../build-test  
hob
```

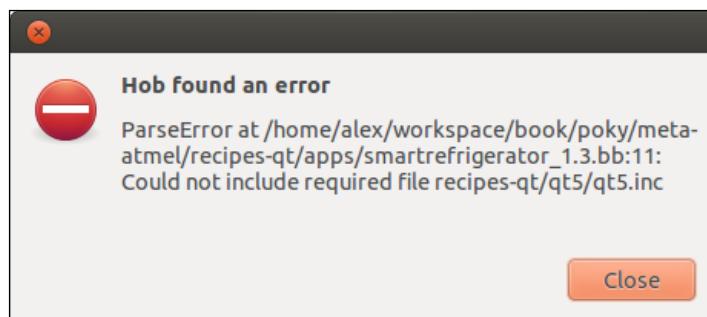
The next step is to establish the layers that are required for your build. You can do this by selecting them in the **Layers** window. The first thing to do for the `meta-atmel` layer is to add it to the build. Although you may start work in an already existing build directory, Hob will not be able to retrieve the existing configurations and will create a new one over the `bblayers.conf` and `local.conf` configuration files. It will mark the added lines using the next `#added by hob` message.



After the corresponding `meta-atmel` layer is added to the build directory, all the supported machines are available in the **Select a machine** drop-down, including those that are added by the `meta-atmel` layer. From the available options, the **sama5d3-xplained** machine needs to be selected:



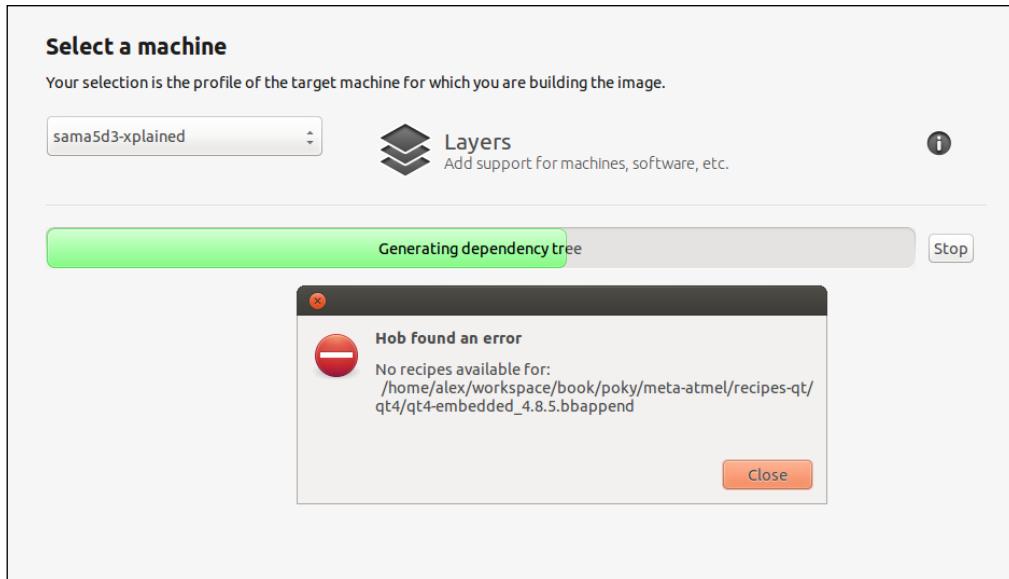
When the Atmel **sama5d3-xplained** machine is selected, an error, shown in the following screenshot, appears:



After adding the `meta-qt5` layer to the layers section, this error disappears and the build process can continue. To retrieve the `meta-qt5` layer, the following git command is necessary:

```
git clone -b dizzy https://github.com/meta-qt5/meta-qt5.git
```

Since all the available configuration files and recipes are parsed, the parsing process takes a while, and after this, you will see an error, as shown in the following screenshot:



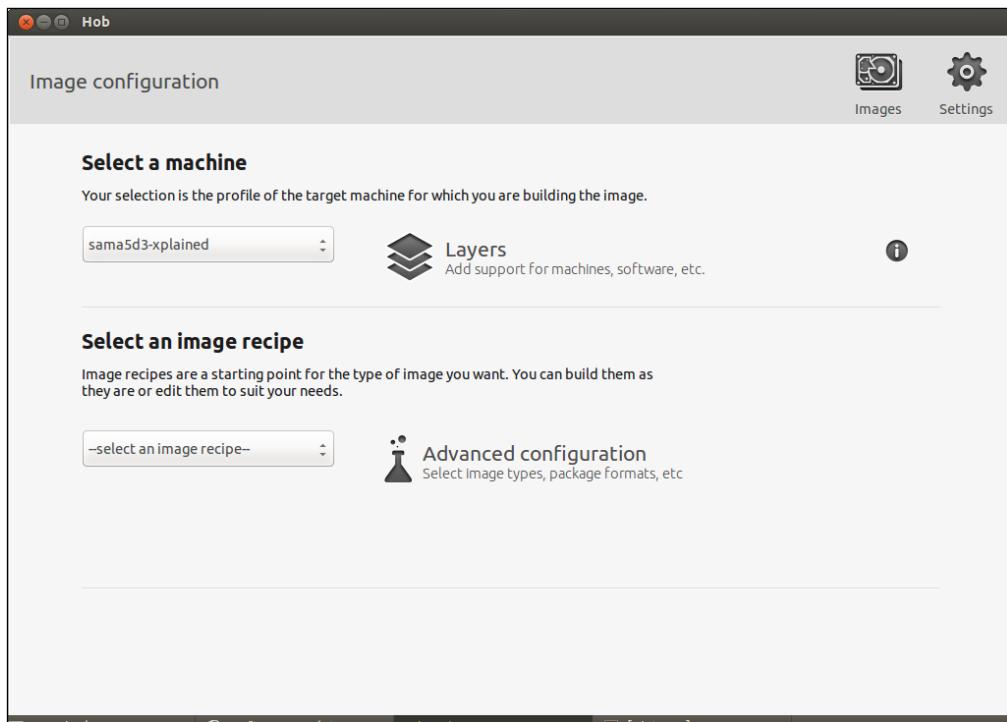
After another quick inspection, you will see the following code:

```
find ../../ -name "qt4-embedded*"
./meta/recipes-qt/qt4/qt4-embedded_4.8.6.bb
./meta/recipes-qt/qt4/qt4-embedded.inc
./meta-atmel/recipes-qt/qt4/qt4-embedded-4.8.5
./meta-atmel/recipes-qt/qt4/qt4-embedded_4.8.5.bbappend
```

The only explanation is the fact the `meta-atmel` layer does not update its recipes but appends them. This can be overcome in two ways. The simplest one would be to update the recipe the `.bbappend` file and make sure that the new available recipe is transformed into a patch for the upstream community. A patch with the required changes inside the `meta-atmel` layer will be explained to you shortly, but first, I will present the available options and the necessary changes that are needed to resolve the problems existing in the build process.

The other solution would be to include the required recipes that `meta-atmel` needs for the build process. The best place for it to be available would be also in `meta-atmel`. However, in this case, the `.bbappend` configuration file should be merged with the recipe, since having a recipe and its appended file in the same place does not make much sense.

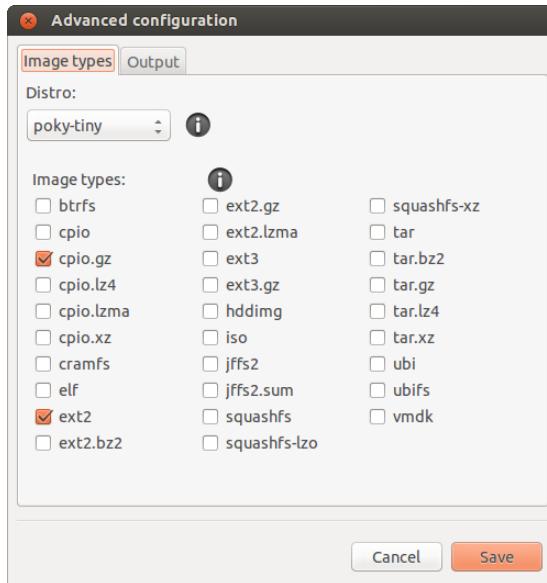
After this problem is fixed, new options will be available to the user, as depicted in the following screenshot:



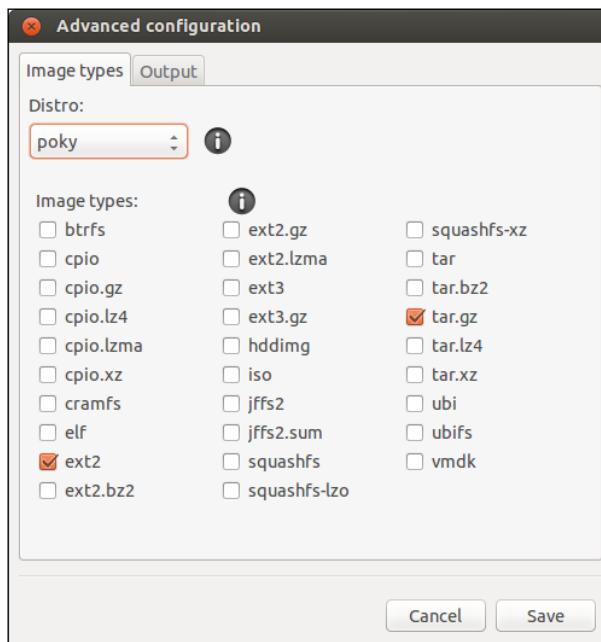
Now, the user has the chance to select the image that needs to be built, as well as the extra configurations that need to be added. These configurations include:

- Selection of the distribution type
- Selection of the image types
- A packaging format
- Other small tweaks around the root filesystem

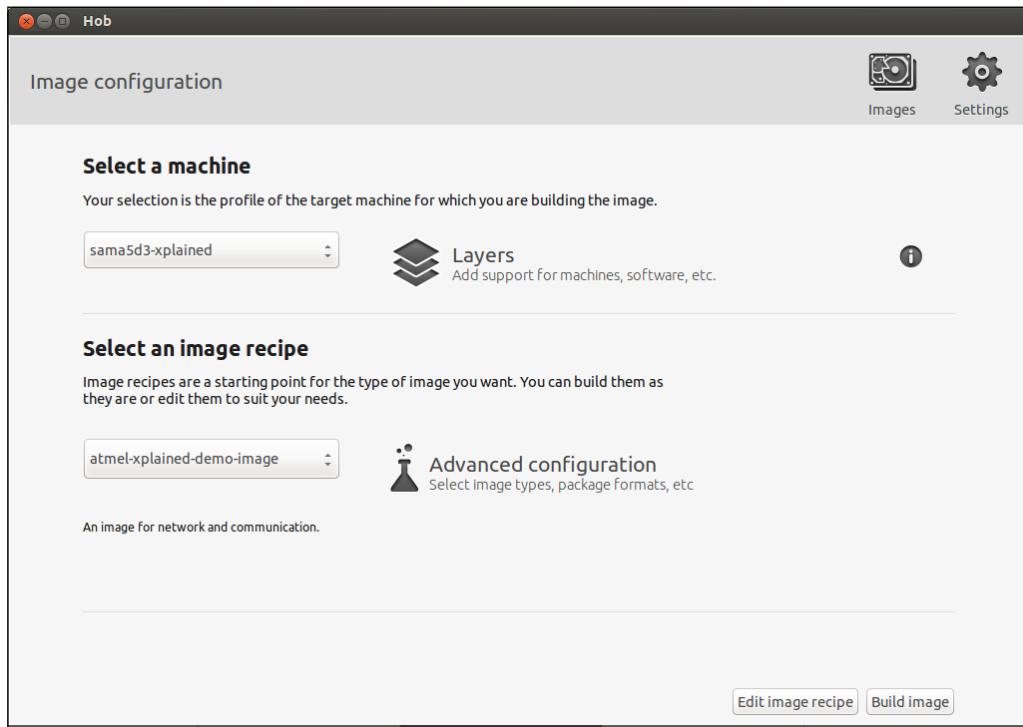
Some of these are depicted in the following screenshot:



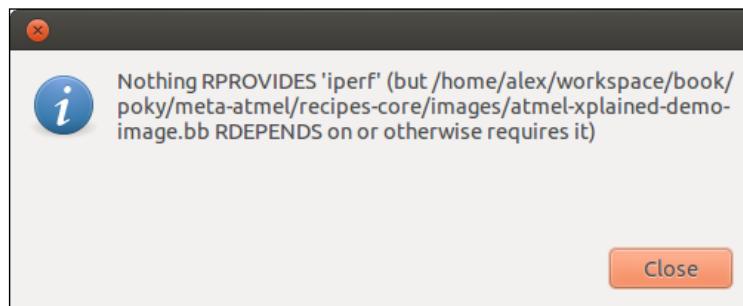
I've chosen to change the distribution type from **poky-tiny** to **poky**, and the resulting root filesystem output format is visible in the following screenshot:



With the tweaks made, the recipes are reparsed, and when this process is finished, the resulting image can be selected so that the build process can start. The image that is selected for this demonstration is the **atmel-xplained-demo-image** image, which corresponds to the recipes with the same name. This information is also displayed in the following screenshot:



The build process is started by clicking on the **Build image** button. A while after the build starts, an error will show up, which tells us that the **meta-atmel** BSP layer requires more of the dependencies that need to be defined by us:



This information is gathered from the `iperf` recipe, which is not available in the included layers; it is available inside the `meta-openembedded/meta-oe` layer. After a more detailed search and update process, there have been a few revelations. There are more layer dependencies than required for the `meta-atmel` BSP layer, which are given as follows:

- The `meta-openembedded/meta-oe` layer
- The `meta-openembedded/meta-networking` layer
- The `meta-openembedded/meta-ruby` layer
- The `meta-openembedded/meta-python` layer
- The `meta-qt5` layer

The end result is available in the `BBLAYERS` variable that is found in the `bblayers.conf` file, shown as follows:

```
#added by hob
BBFILES += "${TOPDIR}/recipes/images/custom/*.bb"
#added by hob
BBFILES += "${TOPDIR}/recipes/images/*.bb"

#added by hob
BBLAYERS = "/home/alex/workspace/book/poky/meta /home/alex/workspace/
book/poky/meta-yocto /home/alex/workspace/book/poky/meta-yocto-bsp /
/home/alex/workspace/book/poky/meta-atmel /home/alex/workspace/book/
poky/meta-qt5 /home/alex/workspace/book/poky/meta-openembedded/meta-
oe /home/alex/workspace/book/poky/meta-openembedded/meta-networking /
/home/alex/workspace/book/poky/meta-openembedded/meta-ruby /home/alex/
workspace/book/poky/meta-openembedded/meta-python"
```

There are some required changes in the `meta-atmel` layer that needs to be made before starting a complete build, given as follows:

- Replace `packagegroup-core-basic` with `packagegroup-core-full-cmdline` because the latest Poky has updated the `packagegroup` names.
- Delete `python-setuptools` because it is not available in the `meta-openembedded/meta-oe` layer anymore, as well as in the new `meta-openembedded/meta-python` layer, which is the new placeholder for all Python-related recipes. The `python-setuptools` tool was removed because it had the ability to download, build, install, upgrade, and uninstall extra Python packages, and is not a mandatory requirement for Yocto. This is its general purpose.
- The preceding change regarding the update to `qt4-embedded-4.8.6` for `qt4-embedded-4.8.5`, as shown earlier, presented errors.

All the changes made to the `meta-atmel` layer are available in following patch:

```
From 35ccf73396da33a641f307f85e6b92d5451dc255 Mon Sep 17 00:00:00
2001
From: "Alexandru.Vaduva" <vaduva.jan.alexandru@gmail.com>
Date: Sat, 31 Jan 2015 23:07:49 +0200
Subject: [meta-atmel] [PATCH] Update support for
atmel-xplained-demo-image
image.

The latest poky contains updates regarding the qt4 version support
and also the packagegroup naming.
Removed packages which are no longer available.

Signed-off-by: Alexandru.Vaduva <vaduva.jan.alexandru@gmail.com>
---
recipes-core/images/atmel-demo-image.inc | 3 ++
...qt-embedded-linux-4.8.4-phonon-colors-fix.patch | 26
-----
...qt-embedded-linux-4.8.4-phonon-colors-fix.patch | 26
+++++
recipes-qt/qt4/qt4-embedded_4.8.5.bbappend | 2 --
recipes-qt/qt4/qt4-embedded_4.8.6.bbappend | 2 ++
5 files changed, 29 insertions(+), 30 deletions(-)
delete mode 100644 recipes-qt/qt4/qt4-embedded-4.8.5/qt-embedded-
linux-4.8.4-phonon-colors-fix.patch
create mode 100644 recipes-qt/qt4/qt4-embedded-4.8.6/qt-embedded-
linux-4.8.4-phonon-colors-fix.patch
delete mode 100644 recipes-qt/qt4/qt4-embedded_4.8.5.bbappend
create mode 100644 recipes-qt/qt4/qt4-embedded_4.8.6.bbappend

diff --git a/recipes-core/images/atmel-demo-image.inc b/recipes-
core/images/atmel-demo-image.inc
index fe13303..a019586 100644
--- a/recipes-core/images/atmel-demo-image.inc
+++ b/recipes-core/images/atmel-demo-image.inc
@@ -2,7 +2,7 @@ IMAGE_FEATURES += "ssh-server-openssh package-
management"

IMAGE_INSTALL = "\\\\
    packagegroup-core-boot \
-   packagegroup-core-basic \
+   packagegroup-core-full-cmdline \
     packagegroup-base-wifi \
     packagegroup-base-bluetooth \\\\

```

```

    packagegroup-base-usbgadget \
@@ -23,7 +23,6 @@ IMAGE_INSTALL = "\"
    python-smbus \
    python-ctypes \
    python-pip \
-   python-setuptools \
    python-pycurl \
    gdbserver \
    usbutils \
diff --git a/recipes-qt/qt4/qt4-embedded-4.8.5/qt-embedded-linux-
4.8.4-phonon-colors-fix.patch b/recipes-qt/qt4/qt4-embedded-
4.8.5/qt-embedded-linux-4.8.4-phonon-colors-fix.patch
deleted file mode 100644
index 0624eef..0000000
--- a/recipes-qt/qt4/qt4-embedded-4.8.5/qt-embedded-linux-4.8.4-
phonon-colors-fix.patch
+++ /dev/null
@@ -1,26 +0,0 @@
-diff --git a/src/3rdparty/phonon/gstreamer/qwidgetvideosink.cpp
b/src/3rdparty/phonon/gstreamer/qwidgetvideosink.cpp
-index 89d5a9d..8508001 100644
---- a/src/3rdparty/phonon/gstreamer/qwidgetvideosink.cpp
-+++ b/src/3rdparty/phonon/gstreamer/qwidgetvideosink.cpp
-@@ -18,6 +18,7 @@
- #include <QApplication>
- #include "videowidget.h"
- #include "qwidgetvideosink.h"
-+#include <gst/video/video.h>
-
- QT_BEGIN_NAMESPACE
-
-@@ -106,11 +107,7 @@ static GstStaticPadTemplate
template_factory_rgb =-      GST_STATIC_PAD_TEMPLATE("sink",-
GST_PAD_SINK,
-
GST_PAD_ALWAYS,
--GST_STATIC_CAPS("video/x-raw-rgb, "
--"framerate =
(fraction) [ 0, MAX ], "
--"width = (int) [ 1,
MAX ], "
--"height = (int) [ 1,
MAX ], "
--"bpp = (int) 32"));
-+GST_STATIC_CAPS(GST_VIDEO_CAPS_XRGB_
HOST_ENDIAN));

```

```
-  
- template <VideoFormat FMT>  
- struct template_factory;  
  
-  
diff --git a/recipes-qt/qt4/qt4-embedded-4.8.6/qt-embedded-linux-  
4.8.4-phonon-colors-fix.patch b/recipes-qt/qt4/qt4-embedded-  
4.8.6/qt-embedded-linux-4.8.4-phonon-colors-fix.patch  
new file mode 100644  
index 0000000..0624eef  
--- /dev/null  
+++ b/recipes-qt/qt4/qt4-embedded-4.8.6/qt-embedded-linux-4.8.4-  
phonon-colors-fix.patch  
@@ -0,0 +1,26 @@  
+diff --git a/src/3rdparty/phonon/gstreamer/qwidgetvideosink.cpp  
b/src/3rdparty/phonon/gstreamer/qwidgetvideosink.cpp  
+index 89d5a9d..8508001 100644  
+--- a/src/3rdparty/phonon/gstreamer/qwidgetvideosink.cpp  
++++ b/src/3rdparty/phonon/gstreamer/qwidgetvideosink.cpp  
+@@ -18,6 +18,7 @@  
+ #include <QApplication>  
+ #include "videowidget.h"  
+ #include "qwidgetvideosink.h"  
++#include <gst/video/video.h>  
+  
+ QT_BEGIN_NAMESPACE  
+  
+@@ -106,11 +107,7 @@ static GstStaticPadTemplate  
template_factory_rgb += GST_STATIC_PAD_TEMPLATE("sink", +  
                                                GST_PAD_SINK, +  
                                                GST_PAD_ALWAYS, +  
                                                GST_STATIC_CAPS("video/x-raw-rgb, "  
+                                                 "framerate = (fraction)  
+- [ 0, MAX ], "  
+--                                                 "width = (int) [ 1,  
MAX ], "  
+--                                                 "height = (int) [ 1,  
MAX ], "  
+--                                                 "bpp = (int) 32");  
++ GST_STATIC_CAPS(GST_VIDEO_CAPS_xRGB_  
HOST_ENDIAN));  
+  
+ template <VideoFormat FMT>  
+ struct template_factory;  
+
```

```

diff --git a/recipes-qt/qt4/qt4-embedded_4.8.5.bbappend b/recipes-
qt/qt4/qt4-embedded_4.8.5.bbappend
deleted file mode 100644
index bbb4d26..0000000
--- a/recipes-qt/qt4/qt4-embedded_4.8.5.bbappend
+++ /dev/null
@@ -1,2 +0,0 @@
-FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}-${PV}:" 
-SRC_URI += "file://qt-embedded-linux-4.8.4-phonon-colors-
fix.patch"
diff --git a/recipes-qt/qt4/qt4-embedded_4.8.6.bbappend b/recipes-
qt/qt4/qt4-embedded_4.8.6.bbappend
new file mode 100644
index 0000000..bbb4d26
--- /dev/null
+++ b/recipes-qt/qt4/qt4-embedded_4.8.6.bbappend
@@ -0,0 +1,2 @@
+FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}-${PV}:" 
+SRC_URI += "file://qt-embedded-linux-4.8.4-phonon-colors-
fix.patch"
--
1.9.1

```

This patch has been given in the chapter as an example for Git interaction and is a necessity when creating a patch that needs to be upstream to the community. At the time of writing this chapter, this patch had not yet been released to the upstream community, so this could be a gift for anyone interested in adding a contribution to the meta-atmel community in particular and the Yocto community in general.

The steps necessary to obtain this patch after the changes have been made, are described shortly. They define the steps needed to generate the patch, as shown in the following command, and is 0001-Update-support-for-atmel-xplained-demo-image-image.patch. It can be upstream to the community or directly to the maintainer of the meta-atmel layer using the information available in the README file and the git send-email command:

```

git status
git add --all .
git commit -s
git fetch -a
git rebase -i origin/master
git format-patch -s --subject-prefix='meta-atmel] [PATCH' origin/master
vim 0001-Update-support-for-atmel-xplained-demo-image-image.patch

```

Toaster

Toaster represents an alternative to Hob, which at a given point in time, will replace it completely. It is also a web-based interface for the BitBake command line. This tool is much more effective than Hob; it is not only able to do the most common tasks in a similar manner as Hob, but it also incorporates a build analysis component that collects data regarding the build process and the resultant outcome. These results are presented in a very easy-to-grasp manner, offering the chance to search, browse, and query the information.

From the collected information, we can mention the following:

- Structure of the image directory
- The available build configurations
- The outcome of a build along with the errors and registered warnings
- The packages present in an image recipe
- Recipes and packages that are built
- Tasks that are executed
- Performance data regarding executed tasks, such as CPU usage, time, and disk I/O usage
- Dependency and reverse dependencies for recipes

There are also some drawbacks to the Hob solution. Toaster does not yet offer the ability to configure and launch a build. However, there are initiatives taken to include these functionalities that Hob has inside Toaster, which will be implemented in the near future.

The current status of the Toaster Project permits the execution in various setups and running modes. Each of them will be presented and accordingly defined as follows:

- **Interactive mode:** This is the mode available and released with the Yocto Project 1.6 release version. It is based on a `toasterui` build recording component and a `toastergui` build inspection and statistics user interface.
- **Managed mode:** In addition to the Yocto Project 1.6 release version, this is the mode that handles build configurations, scheduling, and executions that are triggered from the web interface.
 - **Remote managed mode:** This is a hosted Toaster mode and is defined for production because it offers support for multiple users and customized installations.

- **Local managed mode or _local_ is:** This is the mode available after a Poky checkout and permits running builds using the local machine code and build directory. It is also used by anyone who interacts with a Toaster project for the first time.
- For the **interactive mode**, building with tools, such as AutoBuilder, BuildBot, or Jenkins, a setup separated from the hardware on which the Yocto Project builds are running will be required. Behind a normal instance of Toaster, there are three things that happen:
 - A BitBake server is started
 - A Toaster UI is started and connected to the BitBake server as well as to an SQL database
 - A web server is started for the purpose of reading information related to a database and displaying it on the web interface

There are scenarios when multiple Toaster instances are running on multiple remote machines, or when a single Toaster instance is shared among multiple users and build servers. All of them can be resolved by modifying the mode that the Toaster starts in and changing the SQL database and location of the web server accordingly. By having a common SQL database, a web server, and multiple BitBake servers with the Toaster user interface for each separate build directory, you can solve problems involved in the previously mentioned scenarios. So, each component in a Toaster instance can be run on a different machine, as long as communication is done appropriately and the components know about each other.

To set up an SQL server on a Ubuntu machine, a package needs to be installed, using the following command:

```
apt-get install mysql-server
```

Having the necessary packages is not enough; setting them up is also required. Therefore, the proper username and password for the access web server is necessary, along with the proper administration rights for the MySQL account. Also, a clone of the Toaster master branch would be necessary for the web server, and after the sources are available, make sure that inside the `bitbake/lib/toaster/toastermain/settings.py` file, the `DATABASES` variable indicates the previous setup of the database. Make sure that you use the username and password defined for it.

With the set up done, the database synchronization can begin in the following way:

```
python bitbake/lib/toaster/manage.py syncdb
python bitbake/lib/toaster/manage.py migrate orm
python bitbake/lib/toaster/manage.py migrate bldcontrol
```

Now, the web server can be started using the `python bitbake/lib/toaster/manage.py runserver` command. For background execution, you can use the `nohup python bitbake/lib/toaster/manage.py runserver 2>toaster_web.log >toaster_web.log &` command.

This may be enough for starters, but as case logs are required for the builds, some extra setup is necessary. Inside the `bitbake/lib/toaster/toastermain/settings.py` file, the `DATABASES` variable indicates the SQL database for the logging server. Inside the build directory, call the source `toaster start` command and make sure that the `conf/toaster.conf` file is available. Inside this file, make sure that the Toaster and build history `bbclasses` are enabled to record information about the package:

```
INHERIT += "toaster"  
INHERIT += "buildhistory"  
BUILDHISTORY_COMMIT = "1"
```

After this set up is available, start the BitBake server and the logging interface with these commands:

```
bitbake --postread conf/toaster.conf --server-only -t xmlrpc -B  
localhost:0 && export BB SERVER=localhost:-1  
nohup bitbake --observe-only -u toasterui >toaster_ui.log &
```

After this is done, the normal build process can be started and builds can begin while the build is running inside the web interface logs and data is available to be examined. One quick mention, though: do not forget to kill the BitBake server after you have finished working inside the build directory using the `bitbake -m` command.

The local is very similar to the builds of the Yocto Project presented until now. This is the best mode for individual usage and learning to interact with the tool. Before starting the setup process, a few packages are required to be installed, using the following command lines:

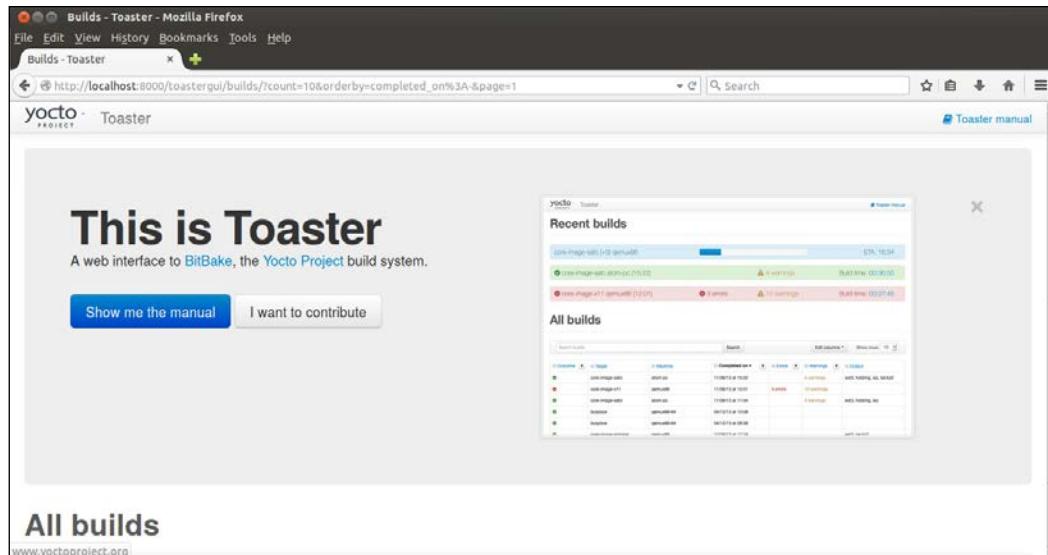
```
sudo apt-get install python-pip python-dev build-essential  
sudo pip install --upgrade pip  
sudo pip install --upgrade virtualenv
```

After these packages are installed, make sure that you install the components required by Toaster; here, I am referring to the Django and South packages:

```
sudo pip install django==1.6
sudo pip install South==0.8.4
```

For interaction with the web server, the 8000 and 8200 ports are necessary, so make sure that they are not already reserved for other interactions. With this in mind, we can start the interaction with Toaster. Using the Poky build directory available from the downloads in the previous chapters, call the oe-init-build-env script to create a new build directory. This can be done on an already existing build directory, but having a new one will help identify the extra configuration files available for interaction with Toaster.

After the build directory is set according to your needs, the source toaster start command should be called, as mentioned previously, to start Toaster. At `http://localhost:8000`, you will see the following screenshot if no build is executed:



Run a build in the console, and it will be automatically updated in the web interface, as shown in the following screenshot:

The screenshot shows a Mozilla Firefox browser window titled "Builds - Toaster". The URL is http://localhost:8000/toastergui/builds/?count=10&orderby=completed_on%3A&page=1. The main content area displays the "Recent builds" section, which lists three completed builds:

Build	Status	Build time
core-image-sato (25 seconds)	Success	00:10:04
core-image-sato-armv6-pc (15:20)	Success	02:36:58
core-image-lt (2:01)	Failure	02:27:48

Below this is the "All builds" section, which shows a table of all build entries. The first entry is highlighted:

Build	Target	Machine	Completed on	Errors	Warnings	Output
core-image-sato	qemux86	qemux86	01/02/15 13:52	0	0	core-image-sato.log
core-image-lt	qemux86	qemux86	01/02/15 02:38	4 errors	4 warnings	core-image-lt.log

At the bottom of the page, there is a status bar with the text "virtual/kernel qemux86 (2 minutes ago)" and "ETA: in 16 minutes from now".

After the build is finished, the web interface will be updated accordingly. I closed the header image and information to make sure that only the builds are visible in the web page.

The screenshot shows the same Mozilla Firefox browser window as the previous one, but the build status has been updated. The "Recent builds" section now shows the completed build as successful:

Build	Status	Build time
virtual/kernel qemux86 (13 minutes ago)	Success	00:17:50

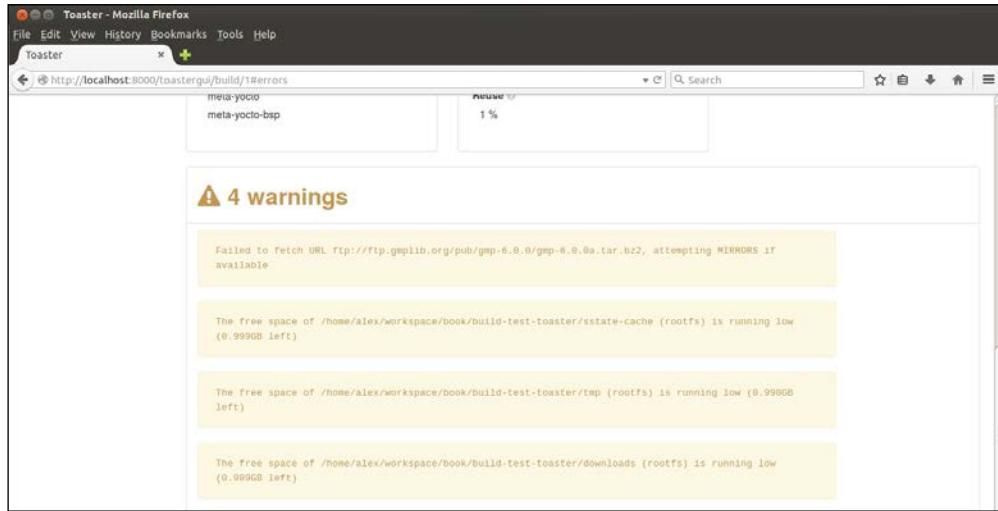
The "All builds" section still shows both builds, but the second build's status is now indicated by a red error icon:

Build	Target	Machine	Completed on	Errors	Warnings	Output
core-image-sato	qemux86	qemux86	01/02/15 13:52	0	0	core-image-sato.log
virtual/kernel	qemux86	qemux86	01/02/15 02:38	4 errors	4 warnings	virtual/kernel.log

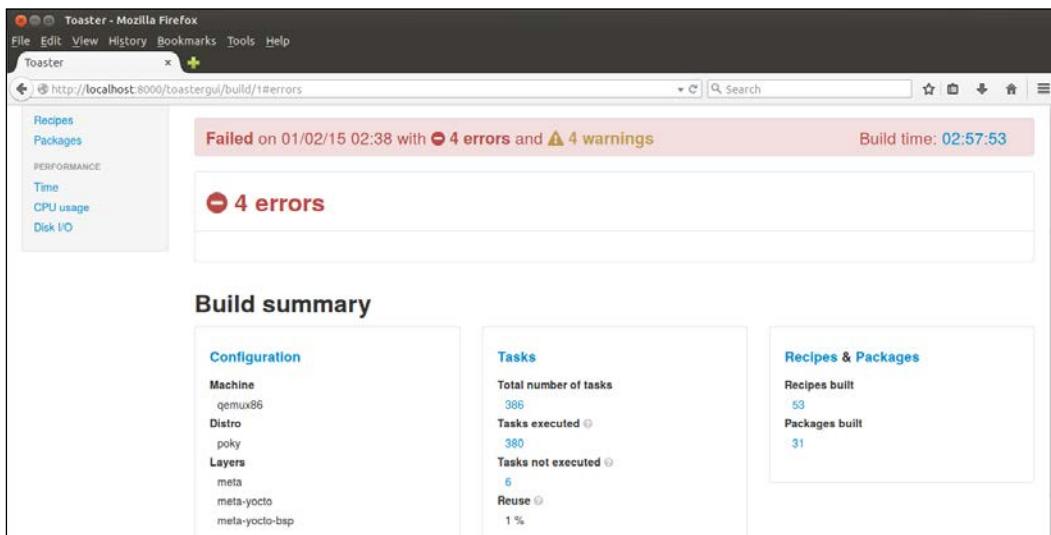
At the bottom of the page, the status bar now shows "virtual/kernel qemux86 (11 hours ago)" and "Build time: 02:57:53".

As seen in the preceding example, there are two builds that have finished in the preceding screenshot. Both of them are kernel builds. The first one finished with success, while the second has some errors and warnings. I did this as an example to present the user with alternative outputs for their build.

The build that failed took place due to lack of memory and space on the host machine, as seen in the following screenshot:



For the failing build, a detailed fail report is available, as displayed in the following screenshot:



The build that finished successfully offers access to a lot of information. The following screenshot shows interesting features that a build should have. It shows, for the kernel build, all the BitBake variables used, their values, their location, and a short description. This information is very useful for all developers, not only because it offers all of this at a single location, but also because it offers a search option that reduces the search time spent looking for a troublesome variable to a minimum:

The screenshot shows a Mozilla Firefox browser window titled "Configvars - Toaster". The URL is "http://localhost:8000/toastergu/build/2/configvars?count=100&orderby=variable_name%3A%2B&filter=description". The main content area displays a table with the heading "304 variables found". The table has columns for Variable, Value, Set in file, and Description. The first three rows of the table are as follows:

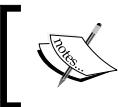
Variable	Value	Set in file	Description
ALLOW_EMPTY_\${PN}-dbg	1	/home/alex/workspace/book/poky/meta/conf/bitbake.conf	Specifies if an output package should still be produced if it is empty.
ALLOW_EMPTY_\${PN}-dev	1	/home/alex/workspace/book/poky/meta/conf/bitbake.conf	Specifies if an output package should still be produced if it is empty.
ASSUME_PROVIDED	bzip2-native chrpath-native git-native grep-native diffstat-native patch-native perl-native-runtime python-native-runtime tar-native virtual/libnl-n...	/home/alex/workspace/book/build-test-toaster/conf/local.conf	List of packages (recipes actually) that are assumed to be implicitly available. BitBake does not build these packages.

To stop Toaster, the source `toaster stop` command can be used after the execution activities are finished.

Inside a build directory, Toaster creates a number of files; their naming and purpose are presented in the following lines:

- `bitbake-cookerdaemon.log`: This log file is necessary for the BitBake server
- `.toastermain.pid`: This is the file that contains pid of the web server
- `.toasterui.pid`: It contains the DSI data bridge, pid
- `toaster.sqlite`: This is the database file
- `toaster_web.log`: This is the web server log file
- `toaster_ui.log`: This is the log file used for components of the user interface

With all of these factors mentioned, let's move to the next component, but not before offering a link to some interesting videos about Toaster.



Information about Toaster Manual 1.7 can be accessed at <https://www.yoctoproject.org/documentation/toaster-manual-17>.

Autobuilder

Autobuilder is the project responsible for QA, and a testing build is available inside the Yocto Project. It is based on the BuildBot project. Although this topic isn't dealt with in this book, for those of you interested in the BuildBot project, you can find more information about it in the following information box.



The starting page of Buildbot can be accessed at <http://trac.buildbot.net/>. You can find a guide on quick starting BuildBot at <http://docs.buildbot.net/0.8.5/tutorial/tour.html>, and its concepts can be found at <http://docs.buildbot.net/latest/manual/concepts.html>.

We are now going to address a software area that is very poorly treated by developers in general. Here, I am referring to the testing and quality assurance of a development process. This is, in fact, an area that requires more attention from us, including me as well. The Yocto Project through the AutoBuilder initiative tries to bring more attention to this area. Also, in the past few years, there has been a shift toward QA and **Continuous Integration (CI)** of available open source projects, and this can primarily be seen in the Linux Foundation umbrella projects.

The Yocto Project is actively involved in the following activities as part of the AutoBuilder project:

- Publishing the testing and QA plans using Bugzilla test cases and plans (<https://bugzilla.yoctoproject.org>).
- Demonstrating these plans and making them accessible for everyone to see. Of course, for this, you will need a corresponding account.
- Developing tools, tests, and QA procedures for everyone to use.

Having the preceding activities as a foundation, they offer access to a public AutoBuilder that shows the current status of the Poky master branch. Nightly builds and test sets are executed for all the supported targets and architectures and are all available for everyone at <http://autobuilder.yoctoproject.org/>.



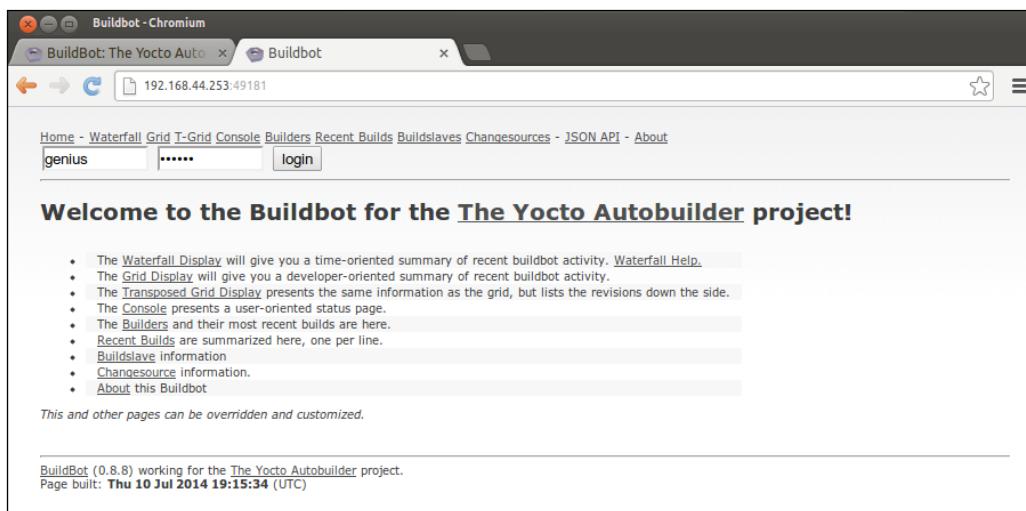
If you do not have a Bugzilla account to access the QA activities done within the Yocto Project, refer to <https://wiki.yoctoproject.org/wiki/QA>.

To interact with the AutoBuilder Project, the setup is defined in the README-QUICKSTART file as a four-step procedure:

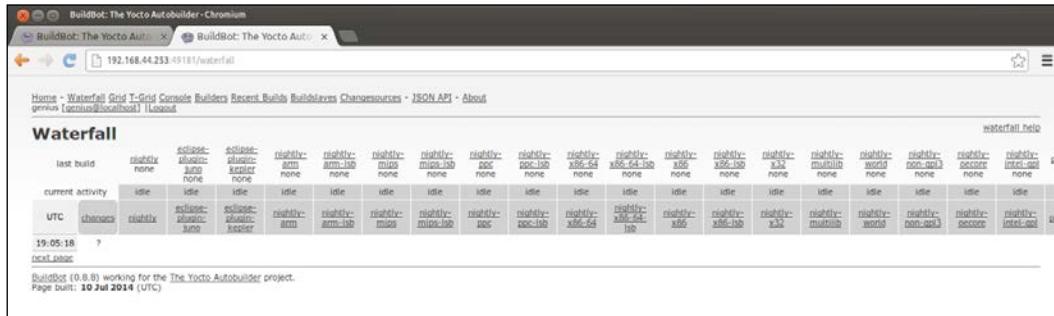
```
cat README-QUICKSTART
Setting up yocto-autobuilder in four easy steps:
-----
git clone git://git.yoctoproject.org/yocto-autobuilder
cd yocto-autobuilder
. ./yocto-autobuilder-setup
yocto-start-autobuilder both
```

The configuration files for this project are available inside the config directory. The autobuilder.conf file is used to define the parameters for the project, such as DL_DIR, SSTATE_DIR, and other build artifacts are very useful for a production setup, though not so useful for a local one. The next configuration file to inspect is yoctoABConfig.py, available in the yocto-controller directory where it defines the properties for the executed builds.

At this point, the AutoBuilder should be running. If it is started inside a web interface, the result should look similar to the following screenshot:



As it can be visible from the header of the web page, there are multiple options available not only for the executed builds, but also for a different view and perspective of them. Here is one of the visualization perspectives:



This project has more to offer to its users, but I will let the rest be discovered through trial and error and a reading of the README file. Keep in mind that this project was built with Buildbot, so the workflow is very similar to it.

Summary

In this chapter, you were presented with a new set of components that are available in the Yocto Project. Here, I am referring to the Hob, Toaster, and AutoBuilder projects. The chapter first introduced Hob as a BitBake alternative. It was followed by the Toaster alternative to Hob, which also comes with a lot of interesting features, and although it is not at its best now, over time, it will become a real solution for developers who are not interested in learning a new technology. Instead, they only interact with a tool to get what they want in a quick and easy manner. This chapter finished with the AutoBuilder project that offers a QA and testing platform for the Yocto Project community and can be transformed in a continuous integration tool.

In the next chapter, some of the other tools will be presented, but this time, the focus will move a little towards the exterior of the community and also its small tools. We will also cover projects and tools, such as Swabber, a project that is continuously in a developing stage. We will also take a look at Wic, a little tool with great personality, and the new sensation from Linaro called LAVA. I hope you enjoy learning all of them.

9

Wic and Other Tools

In this chapter, you will be given a brief introduction to a number of tools that address various problems and solves them in ingenious ways. This chapter can be thought of as an appetizer for you. If any of the tools presented here seem to interest you, I encourage you to feed your curiosity and try to find more about that particular tool. Of course, this piece of advice applies to any information presented in this book. However, this bit of advice holds true particularly for this chapter because I've chosen a more general description for the tools I've presented. I've done this as I've assumed that some of you may not be interested in lengthy descriptions and would only want to focus your interest in the development process, rather than in other areas. For the rest of you who are interested in finding out more about other key areas, please feel free to go through the extensions of information available throughout the chapter.

In this chapter, a more detailed explanation of components, such as Swabber, Wic, and LAVA, will be offered. These tools are not the ones, which an embedded developer will encounter on everyday jobs, though interaction with such tools could make life a little easier. The first thing I should mention about these tools is that they have nothing in common with each other, and are very different from each other and address different requests. If Swabber, the first tool presented here, is used for access detection on a host development machine, the second tool represents a solution to the limitations that BitBake has with complex packaging options. Here, I am referring to the wic tool. The last element presented in this chapter is the automation testing framework called LAVA. It is an initiative from Linaro, a project that, in my opinion, is very interesting to watch. They are also combined with a continuous integration tool, like Jenkins, and this could make it a killer combination for every taste.

Swabber

Swabber is a project, which although is presented on Yocto Project's official page, is said to be a work in progress; no activity has been done on it since September 18, 2011. It does not have a maintainers file where you can find more information about its creators. However, the committers list should be enough for anyone interested in taking a deeper look at this project.

This tool was selected for a short introduction in this chapter because it constitutes another point of view of the Yocto Project's ecosystem. Of course, a mechanism for access detection into the host system is not a bad idea and is very useful to detect accesses that could be problematic for your system, but it is not the first tool that comes to mind when developing software. When you have the possibility of redoing your build and inspecting your host ecosystem manually, you tend to lose sight of the fact that tools could be available for this task too, and that they could make your life easier.

For interaction with Swabber, the repository needs to be cloned first. The following command can be used for this purpose:

```
git clone http://git.yoctoproject.org/git/swabber
```

After the source code is available on the host, the content of the repository should look as follows:

```
tree swabber/
swabber/
├── BUGS
├── canonicalize.c
├── canonicalize.h
├── COPYING
├── detect_distro
└── distros
    ├── Fedora
    │   └── whitelist
    ├── generic
    │   ├── blacklist
    │   ├── filters
    │   └── whitelist
    ├── Ubuntu
    │   └── blacklist
```

```

|   |   ├── filters
|   |   └── whitelist
|   └── Windriver
|       └── whitelist
├── dump_blob.c
└── lists.c
    └── lists.h
    └── load_distro.c
    └── Makefile
    └── packages.h
    └── README
    └── swabber.c
    └── swabber.h
    └── swabprof.c
    └── swabprof.in
    └── swab_testf.c
    └── update_distro
    └── wandering.c
        └── wandering.h

```

5 directories, 28 files

As you can see, this project is not a major one, but consists of a number of tools made available by a passionate few. This includes two guys from **Windriver**: Alex deVries and David Borman. They worked on their own on the previously presented tools and made them available for the open source community to use. Swabber is written using the C language, which is a big shift from the usual Python/Bash tools and other projects that are offered by the Yocto Project community. Every tool has its own purpose, the similitude being that all the tools are built using the same Makefile. Of course, this isn't restricted only to the usage of binaries; there are also two bash scripts available for distribution detect and update.

More information about the tool can be found from its creators. Their e-mail addresses, which are available in the commits for the project, are alex.devries@windriver.com and david.borman@windriver.com. However, please note that these are the workplace e-mail IDs and the people that worked on Swabber may not have the same e-mail address at the moment.

The interaction with the Swabber tools is well described in the `README` file. Here, information regarding the setup and running of Swabber is available, though, for your sake, this will also be presented in the next few lines, so that you can understand quicker and in an easier manner.

The first required step is the compilation of sources. This is done by invoking the `make` command. After the source code is built and the executables are available, the host distribution can be profiled using the `update_distro` command, followed by the location of the distribution directory. The name we've chosen for it is `Ubuntu-distro-test`, and it is specific for the host distribution on which the tool is executed. This generation process can take some time at first, but after this, any changes to the host system will be detected and the process will take lesser time. At the end of the profiling process, this is how the content of the `Ubuntu-distro-test` directory looks:

```
Ubuntu-distro-test/
├── distro
├── distro.blob
├── md5
└── packages
```

After the host distribution is profiled, a Swabber report can be generated based on the profile created. Also, before creating the report, a profile log can be created for later use along with the reporting process. To generate the report, we will create a log file location with some specific log information. After the logs are available, the reports can be generated:

```
strace -o logs/Ubuntu-distro-test-logs.log -e trace=open,execve -f pwd
./swabber -v -v -c all -l logs/ -o required.txt -r extra.txt -d Ubuntu-
distro-test/ ~ /tmp/
```

This information was required by the tool, as shown in its help information:

```
Usage: swabber [-v] [-v] [-a] [-e]
                -l <logpath> ] -o <outputfile> <filter dir 1> <filter dir 2> ...
Options:
    -v: verbose, use -v -v for more detail
    -a: print progress (not implemented)
    -l <logfile>: strace logfile or directory of log files to read
```

```
-d <distro_dir>: distro directory
-n <distro_name>: force the name of the distribution
-r <report filename>: where to dump extra data (leave empty for
stdout)
-t <global_tag>: use one tag for all packages
-o <outputfile>: file to write output to
-p <project_dir>: directory were the build is being done
-f <filter_dir>: directory where to find filters for whitelist,
    blacklist, filters
-c <task1>,<task2>....: perform various tasks, choose from:
    error_codes: show report of files whose access returned an
error
    whitelist: remove packages that are in the whitelist
    blacklist: highlight packages that are in the blacklist as
        being dangerous
    file_detail: add file-level detail when listing packages
    not_in_distro: list host files that are not in the package
        database
    wandering: check for the case where the build searches for a
        file on the host, then finds it in the project.
    all: all the above
```

From the help information attached in the preceding code, the role of the arguments selected for the test command can be investigated. Also, an inspection of the tool's source code is recommended due to the fact that there are no more than 1550 lines in a C file, the biggest one being the swabber.c file.

The required.txt file contains the information about the packages used and also about the packages specific files. More information regarding configurations is also available inside the extra.txt file. Such information includes files and packages that can be accessed, various warnings and files that are not available in the host database, and various errors and files that are considered dangerous.

For the command on which the tracing is done, the output information is not much. It has only been offered as an example; I encourage you to try various scenarios and familiarize yourselves with the tool. It could prove helpful to you later.

Wic

Wic is a command line tool that can be also seen as an extension of the BitBake build system. It was developed due to the need of having a partitioning mechanism and a description language. As it can be concluded easily, BitBake lacks in these areas and although initiatives were taken to make sure that such a functionality would be available inside the BitBake build system, this was only possible to an extent; for more complex tasks, Wic can be an alternative solution.

In the following lines, I will try to describe the problem associated with BitBake's lack of functionality and how Wic can solve this problem in an easy manner. I will also show you how this tool was born and what source of inspiration source was.

When an image is being built using BitBake, the work is done inside an image recipe that inherits `image.bbclass` for a description of its functionality. Inside this class, the `do_rootfs()` task is the one that the OS responsible for the creation of the root filesystem directory that will be later be included in the final package and includes all the sources necessary to boot a Linux image on various boards. With the `do_rootfs()` task finished, a number of commands are interrogated to generate an output for each one of the image defined types. The definition of the image type is done through the `IMAGE_FSTYPE` variable and for each image output type, there is an `IMAGE_CMD_type` variable defined as an extra type that is inherited from an external layer or a base type described in the `image_types.bbclass` file.

The commands behind every one of these types are, in fact, a shell command-specific for a defined root filesystem format. The best example of this is the `ext3` format. For this, the `IMAGE_CMD_ext3` variable is defined and these commands are invoked, shown as follows:

```
genext2fs -b ${ROOTFS_SIZE} ... ${IMAGE_NAME}.rootfs.ext3  
tune2fs -j ${DEPLOY_DIR_IMAGE}/${IMAGE_NAME}.rootfs.ext3
```

After the commands are called, the output is in the form of a `image-* .ext3` file. It is a newly created EXT3 filesystem according to the `FSTYPES` defined variable value, and it incorporates the root filesystem content. This example presents a very common and basic filesystem creation of commands. Of course, more complex options could be required in an industry environment, options that incorporate more than the root filesystem and add an extra kernel or even the bootloader alongside it, for instance. For these complex options, extensive mechanisms or tools are necessary.

The available mechanism implemented in the Yocto Project is visible inside the `image_types.bbclass` file through the `IMAGE_CMD_type` variable and has this form:

```
image_types_foo.bbclass:  
    IMAGE_CMD_bar = "some shell commands"  
    IMAGE_CMD_baz = "some more shell commands"
```

To use the newly defined image formats, the machine configuration needs to be updated accordingly, using the following commands:

```
foo-default-settings.inc  
    IMAGE_CLASSES += "image_types_foo"
```

By using the `inherit ${IMAGE_CLASSES}` command inside the `image.bbclass` file, the newly defined `image_types_foo.bbclass` file's functionality is visible and ready to be used and added to the `IMAGE_FSTYPE` variable.

The preceding implementation implies that for each implemented filesystem, a series of commands are invoked. This is a good and simple method for a very simple filesystem format. However, for more complex ones, a language would be required to define the format, its state, and in general, the properties of the image format. Various other complex image format options, such as **vmdk**, **live**, and **directdisk** file types, are available inside Poky. They all define a multistage image formatting process.

To use the **vmdk** image format, a `vmdk` value needs to be defined in the `IMAGE_FSTYPE` variable. However, for this image format to be generated and recognized, the `image-vmdk.bbclass` file's functionalities should be available and inherited. With the functionalities available, three things can happen:

- An EXT3 image format dependency is created on the `do_rootfs()` task to make sure the `ext3` image format is generated first. The `vmdk` image format depends on this.
- The `ROOTFS` variable is set for the `boot-directdisk` functionality.
- The `boot-directdisk.bbclass` is inherited.

This functionality offers the possibility of generating images that can be copied onto a hard disk. At the base of it, the `syslinux` configuration file can be generated, and two partitions are also required for the boot up process. The end result consists of an MBR and partition table section followed by a FAT16 partition containing the boot files, `SYSLINUX` and the Linux kernel, and an EXT3 partition for the root filesystem location. This image format is also responsible for moving the Linux kernel, the `syslinux.cfg`, and `ldlinux.sys` configurations on the first partition, and copying using the `dd` command the EXT3 image format onto the second partition. At the end of this process, space is reserved for the root with the `tune2fs` command.

Historically, the usage of `directdisk` was hardcoded in its first versions. For every image recipe, there was a similar implementation that mirrored the basic one and hardcoded the heritage inside the recipe for the `image.bbclass` functionality. In the case of the `vmdk` image format, the `inherit boot-directdisk` line is added.

With regard to custom-defined image filesystem types, one such example can be found inside the `meta-fsl-arm` layer; this example is available inside the `imx23evk.conf` machine definition. This machine adds the next two image filesystem types: `uboot.mxsboot-sdcard` and `sdcard`.

```
meta-fsl-arm/imx23evk.conf
  include conf/machine/include/mxs-base.inc
  SDCARD_ROOTFS ?= "${DEPLOY_DIR_IMAGE}/${IMAGE_NAME}.rootfs.ext3"
  IMAGE_FSTYPES ?= "tar.bz2 ext3 uboot.mxsboot-sdcard sdcard"
```

The `mxs-base.inc` file included in the preceding lines is in return including the `conf/machine/include/fsl-default-settings.inc` file, which in return adds the `IMAGE_CLASSES += "image_types_fsl"` line as presented in the general case. Using the preceding lines offers the possibility for the `IMAGE_CMD` commands to be first executed for the commands available for the `uboot.mxsboot-sdcard` format, followed by the `sdcard IMAGE_CMD` commands-specific image format.

The `image_types_fsl.bbclass` file defines the `IMAGE_CMD` commands, as follows:

```
inherit image_types
  IMAGE_CMD_uboot.mxsboot-sdcard = "mxsboot sd ${DEPLOY_DIR_IMAGE}/u-
boot-${MACHINE}.${UBOOT_SUFFIX} \
${DEPLOY_DIR_IMAGE}/${IMAGE_NAME}.rootfs.uboot.mxsboot-sdcard"
```

At the end of the execution process, the `uboot.mxsboot-sdcard` command is called using the `mxsboot` command. Following the execution of this command, the `IMAGE_CMD_sdcard` specific commands are called to calculate the SD card size and alignment, as well as to initialize the deploy space and set the appropriate partition type to the `0x53` value and copy the root filesystem onto it. At the end of the process, several partitions are available and they have corresponding twiddles that are used to package bootable images.

There are multiple methods to create various filesystems and they are spread over a large number of existing Yocto layers with some documentation available for the general public. There are even a number of scripts used to create a suitable filesystem for a developer's needs. One such example is the `scripts/contrib/mkefidisk.sh` script. It is used to create an EFI-bootable direct disk image from another image format, that is, a `live.hddimg` one. However, a main idea remains: this kind of activity should be done without any middle image filesystem that is generated in intermediary phases and with something other than a partition language that is unable to handle complicated scenarios.

Keeping this information in mind, it seems that in the preceding example, we should have used another script. Considering the fact that it is possible to build an image from within the build system and also outside of it, the search for a number of tools that fit our needs was started. This search ended at the Fedora kickstart project. Although it has a syntax that is also suitable for areas involving deployment efforts, it is often considered to be of most help to developers.



You can find more information about the Fedora Kickstart project at <http://fedoraproject.org/wiki/Anaconda/Kickstart>.



From this project, the most used and interesting components were `clearpart`, `part`, and `bootloader`, and these are useful for our purposes as well. When you take a look at the Yocto Project's Wic tool, it is also available inside the configuration files. If the configuration file for Wic is defined as `.wks` inside the Fedora kickstart project, the configuration file read uses the `.yks` extension. One such configuration file is defined as follows:

```
def pre():
    free-form python or named 'plugin' commands

    clearpart commands
    part commands
    bootloader commands
    named 'plugin' commands

def post():
    free-form python or named 'plugin' commands
```

The idea behind the preceding script is very simple: the `clearpart` component is used to clear the disk of any partitions while the `part` component is used for the reverse, that is, the components used for creating and installing the filesystem. The third tool that is defined is the `bootloader` component, which is used for installation of the bootloader, and also handles the corresponding information received from the `part` component. It also makes sure that the boot process is done as described inside the configuration file. The functions defined as `pre()` and `post()` are used for pre and post calculus for creation of the image, stage image artefacts, or other complex tasks.

As shown in the preceding description, the interaction with the Fedora kickstarter project was very productive and interesting, but the source code is written using Python inside the Wic project. This is due to the fact that a Python implementation for a similar tool was searched for and it was found under the form of the `pykickstarted` library. This is not all that the preceding library was used for by the Meego project inside its **Meego Image Creator (MIC)** tool. This tool was used for a Meego-specific image creation process. Later, this project was inherited by the Tizen project.



For more about MIC, refer to <https://github.com/01org/mic>.



Wic, the tool that I promised to present in this section is derived from the MIC project and both of them use the kickstarter project, so all three are based on plugins that define the behavior of the process of creating various image formats. In the first implementation of Wic, it was mostly a functionality of the MIC project. Here, I am referring to the Python classes it defines that were almost entirely copied inside Poky. However, over time, the project started to have its own implementations, and also its own personality. From version 1.7 of the Poky repository, no direct reference to MIC Python defined classes remained, making Wic a standalone project that had its own defined plugins and implementations. Here is how you can inspect the various configuration of formats accessible inside Wic:

```
tree scripts/lib/image/canned-wks/
scripts/lib/image/canned-wks/
├── directdisk.wks
├── mkefidisk.wks
├── mkgummidisk.wks
└── sdimage-bootpart.wks
```

There are configurations defined inside Wic. However, considering the fact that the interest in this tool has grown in the last few years, we can only hope that the number of supported configurations will increase.

I mentioned previously that the MIC and Fedora kickstarter project dependencies were removed, but a quick search inside the Poky `scripts/lib/wic` directory will reveal otherwise. This is because Wic and MIC are both have the same foundation, the `pykickstarted` library. Though Wic is now heavily based on MIC and both have the same parent, the kickstarter project, their implementations, functionalities, and various configurations make them different entities, which although related have taken different paths of development.

LAVA

LAVA (Linaro Automation and Validation Architecture) is a continuous integration system that concentrates on a physical target or virtual hardware deployment where a series of tests are executed. The executed tests are of a large variety from the simplest ones which only requires booting a target to some very complex scenarios that require external hardware interaction.

LAVA represents a collection of components that are used for automated validation. The main idea behind the LAVA stack is to create a quality controlled testing and automation environment that is suitable for projects of all sizes. For a closer look at a LAVA instance, the reader could inspect an already created one, the official production instance of which is hosted by Linaro in Cambridge. You can access it at <https://validation.linaro.org/>. I hope you enjoy working with it.

The LAVA framework offers support for the following functionalities:

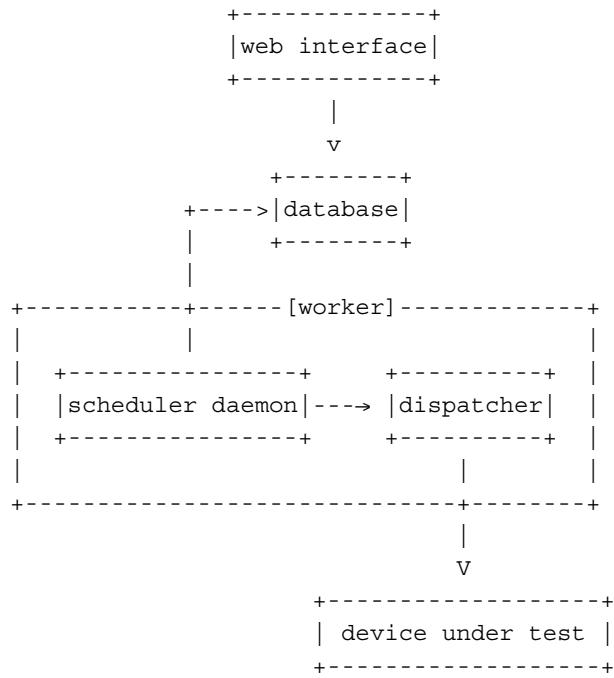
- It supports scheduled automatic testing for multiple packages on various hardware packages
- It makes sure that after a device crashes, the system restarts automatically
- It conducts regression testing
- It conducts continuous integration testing
- It conducts platform enablement testing
- It provides support for both local and cloud solutions
- It provides support for result bundles
- It provides measurements for performance and power consumption

LAVA is primarily written using Python, which is no different from what the Yocto Project offers us. As seen in the Toaster Project, LAVA also uses the Django framework for a web interface and the project is hosted using the Git versioning system. This is no surprise since we are talking about Linaro, a not-for-profit organization that works on free and open source projects. Therefore, the thumb rule applied to all the changes made to the project should return in the upstream project, making the project a little easier to maintain. However, it is also more robust and has better performance.



For those of you interested in more details about how this project can be used, refer to <https://validation.linaro.org/static/docs/overview.html>.

For testing with the LAVA framework, the first step would be to understand its architecture. Knowing this helps not only with test definitions, but also with extending them, as well as the development of the overall project. The major components of this project are as follows:



The first component, the **web interface**, is responsible for user interaction. It is used to store data and submitted jobs using RDBMS, and is also responsible to display the results, device navigation, or as job submission receiver activities that are done through the XMLRPC API. Another important component is represented by the **scheduler daemon**, which is responsible for the allocation of jobs. Its activity is quite simple. It is responsible for pooling the data from a database and reserving devices for jobs that are offered to them by the dispatcher, another important component. The **dispatcher** is the component responsible for running actual jobs on the devices. It also manages the communication with a device, download images, and collects results.

There are scenarios when only the dispatcher can be used; these scenarios involve the usage of a local test or a testing feature development. There are also scenarios where all the components run on the same machine, such as a single deployment server. Of course, the desired scenario is to have components decoupled, the server on one machine, database on another one, and the scheduler daemon and dispatcher on a separate machine.

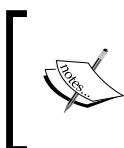
For the development process with LAVA, the recommended host machines are Debian and Ubuntu. The Linaro development team working with LAVA prefer the Debian distribution, but it can work well on an Ubuntu machine as well. There are a few things that need to be mentioned: for the Ubuntu machine, make sure that the universe repositories are available and visible by your package manager.

The first package that is necessary is `lava-dev`; it also has scripts that indicate the necessary package dependencies to assure the LAVA working environment. Here are the necessary commands required to do this:

```
sudo apt-get install lava-dev
git clone http://git.linaro.org/git/lava/lava-server.git
cd lava-server
/usr/share/lava-server/debian-dev-build.sh lava-server

git clone http://git.linaro.org/git/lava/lava-dispatcher.git
cd lava-dispatcher
/usr/share/lava-server/debian-dev-build.sh lava-dispatcher
```

Taking into consideration the location of the changes, various actions are required. For example, for a change in the `templates` directory's HTML content, refreshing the browser will suffice, but any changes made in the `*_app` directory's Python implementation will require a restart of the `apache2ctl` HTTP server. Also, any change made in the `*_daemon` directory's Python sources will require a restart of `lava-server` altogether.

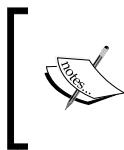


For all of you interested in acquiring more information about LAVA development, the development guide constitutes a good resource of documentation, which is available at <https://validation.linaro.org/static/docs/#developer-guides>.

To install LAVA or any LAVA-related packages on a 64-bit Ubuntu 14.04 machine, new package dependencies are required in addition to the enabled support for universal repositories `deb http://people.linaro.org/~neil.williams/lava jessie main`, besides the installation process described previously for the Debian distribution. I must mention that when the `lava-dev` package is installed, the user will be prompted to a menu that indicates `nullmailer mailname`. I've chosen to let the default one remain, which is actually the host name of the computer running the `nullmailer` service. I've also kept the same configuration defined by default for `smarthost` and the installation process has continued. The following are the commands necessary to install LAVA on a Ubuntu 14.04 machine:

```
sudo add-apt-repository "deb http://archive.ubuntu.com/ubuntu $(lsb_
release -sc) universe"
sudo apt-get update
sudo add-apt-repository "deb http://people.linaro.org/~neil.williams/lava
jessie main"
sudo apt-get update

sudo apt-get install postgresql
sudo apt-get install lava
sudo a2dissite 000-default
sudo a2ensite lava-server.conf
sudo service apache2 restart
```



Information about the LAVA installation process is available at https://validation.linaro.org/static/docs/installing_on_debian.html#. Here, you also find the installation processes for both Debian and Ubuntu distributions.

Summary

In this chapter, you were presented a new set of tools. I will honestly admit that these tools are not the ones used most often in an embedded environment, but they've been introduced in order to offer another point of view to the embedded development environment. This chapter tried to explain to developers that there is more to the embedded world than just development and the tools that help with these tasks. In most cases, the adjacent components are the ones that could inspire and influence the development process the most.

In the next chapter, a short presentation of the Linux real-time requirements and solutions will be presented. We will emphasize the various features that work alongside Linux in this area. A short presentation of the meta-realtime layer will be offered, and features, such as Preempt-RT and NOHZ, will be discussed. Without further ado, let's proceed to the next chapter. I hope you will enjoy its content.

10

Real-time

In this chapter, you will be presented with information on the real-time component of the Yocto Project. Also, in the same context, a short discussion regarding the general purpose of an operating system and a real-time operating system will be explained. We will then move toward the PREEMPT_RT patches that try to change normal Linux into a full powered real-time operating system; we will try to look at it from more angles and at the end, sum it up and draw a conclusion out of it. This is not all, any real-time operation needs its applications, so a short presentation on the do's and don'ts of application writing that is suitable in the context of a real-time operating system, will also be presented. Keeping all of this in mind, I believe it's time to proceed with this chapter content; I hope you enjoy it.

You will find a more detailed explanation of real-time components in this chapter. Also, the relation between Linux and real-time will be shown to you. As everyone knows already, the Linux operation system was designed as a general purpose OS very similar to the already available UNIX. It is very easy to see the fact that a multiuser system, such as Linux, and a real-time one are somewhat in conflict. The main reason for this is that for a general purpose, multiple user operating systems, such as Linux, are configured to obtain a maximal average throughput. This sacrifices latencies that offer exactly the opposite requirements for a real-time operating system.

The definition for real time is fairly easy to understand. The main idea behind it in computing is that a computer or any embedded device is able to offer feedback to its environment in time. This is very different from being fast; it is, in fact, fast enough in the context of a system and fast enough is different for the automobile industry or nuclear power plants. Also, this kind of a system will offer reliable responses to take decisions that don't affect any exterior system. For example, in a nuclear power plant, it should detect and prevent any abnormal conditions to ensure that a catastrophe is avoided.

Understanding GPOS and RTOS

When Linux is mentioned, usually **General Purpose Operating System (GPOS)** is related to it, but over time, the need to have the same benefits as **Real-Time Operating System (RTOS)** for Linux has become more stringent. The challenge for any real-time system is to meet the given timing constraints in spite of the number and type of random asynchronous events. This is no simple task and an extensive number of papers and researches were done on theory of the real-time systems. Another challenge for a real-time system would be to have an upper limit on latency, called a scheduling deadline. Depending on how systems meet this challenge, they can be split into hard, firm, and soft:

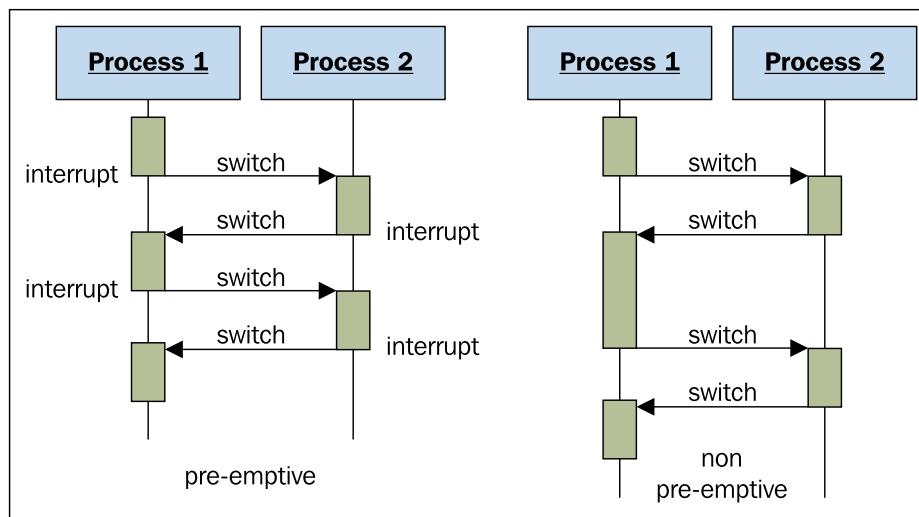
- **Hard real-time system:** This represents system for which a deadline miss will result in a complete system failure.
- **Firm real-time system:** This represents systems for which a deadline miss is acceptable but the system quality can be degraded. Also, after the deadline is missed, the result that is offered is not useful anymore.
- **Soft real-time system:** This represents systems for which missing of deadlines degrades the usefulness of the received result and consequently, of the quality of the system. In these kind of systems, the meeting of the deadline is seen as a goal than as a strict requirement.

There are multiple reasons for Linux not being suitable as a RTOS:

- **Paging:** The page swap process through virtual memory is without limits. There is no method in place to know the time that will pass until you can get a page from a disk, and this implies that there is no upper limit to the delay caused by the fault in a page.
- **Coarsed-grained synchronization:** Here, the definition of the Linux kernel is not preemptible. This means that once a process is inside the kernel context, it cannot be preempted until it exits the context. At an event occurrence, the new event needs to wait for scheduling until the already available one exits the kernel context.
- **Batching:** An operation can be batched for a more efficient use of resources. The simplest example of this is the page freeing process. Instead of freeing each separate page, Linux is able to pass multiple pages and clean as many as possible.
- **Request reordering:** The I/O requests can be reordered for processes, making the process of using hardware more efficient.

- **Fairness in scheduling:** This is a UNIX heritage and refers to the fact that a scheduler tries to be fair with all running processes. This property offers the possibility of lower priority processes that have been waiting for a long time to be scheduled before higher priority ones.

All the preceding characteristics constitute the reason why an upper boundary cannot be applied to the latency of a task or process, and also why Linux cannot become a hard real-time operating system. Let's take a look at the following diagram which illustrates the approaches of Linux OS to offer real-time characteristics:



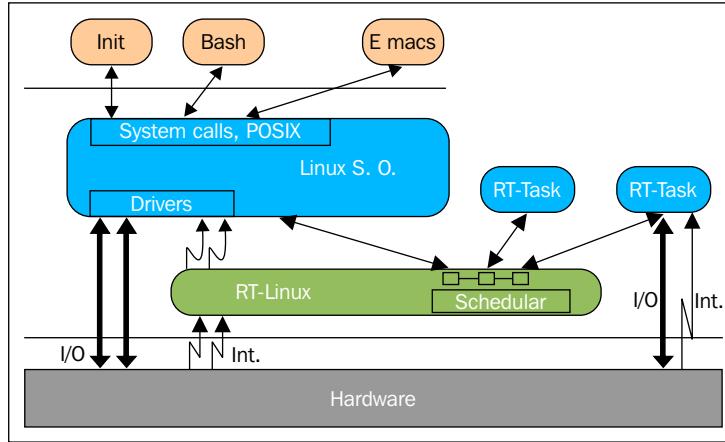
The first thing anyone can do to improve the latency of the standard Linux operating system would be to try and make a change to the scheduling policies. The default Linux time sharing scheduling policies are called **SCHED_OTHER**, and they use a fairness algorithm, giving all processes zero priority, the lowest one available. Other such scheduling policies are **SCHED_BATCH** for batch scheduling of the processes and the **SCHED_IDLE**, which is suitable for the scheduling of extremely low priority jobs. The alternatives to this scheduling policy are **SCHED_FIFO** and **SCHED_RR**. Both of them are intended as real-time policies and are time-critical applications that require precise control processes and their latencies.

To offer more real-time characteristics to a Linux operating system, there are also two more approaches that can be presented. The first one refers to a more preemptive implementation of the Linux kernel. This approach can take advantage of the already available spinlock mechanism used for SMP support, making sure that multiple processes are prevented from executing simultaneously, though in the context of a single processor, the spinlocks are no ops. The interrupt handling also requires modifications this rescheduling to make possible if another higher priority process appears; in this situation, a new scheduler might also be required. This approach offers the advantage of not changing the interaction of a user space and the advantage of using APIs, such as POSIX or others. The drawback of this is that the kernel changes are very serious and every time a kernel version changes, these changes need to be adapted accordingly. If this work was not enough already, the end result is not fully real-time operating system, but one that reduces the latency of the operating system.

The other available implementation is interrupt abstraction. This approach is based on the fact that not all systems require a hard real-time determinism and most of them only require a section of their task to be executed in a real-time context. The idea behind this approach is to run Linux with the priority of an idle task under a real-time kernel and non-real-time tasks to continue to execute them as they normally do. This implementation fakes the disabling of an interrupt for the real-time kernel, but in fact, it is passed to the real-time kernel. For this type of implementation, there are three available solutions:

- **RTLinux:** It represents the original implementation of the interrupt abstraction approach and was developed at the Institute of Mining and Technology, New Mexico. Although it still has an open source implementation, most of the development is now done through FSMLabs engineers, later required by the Wind River System on the commercial version of it. The commercial support for RTLinux ended in August 2011.
- **RTAI:** It is an enhancement made to the RTLinux solution developed in the department of Aerospace Engineering from the Politecnico di Milano. This project is a very active with a high number of developers and has current releases available.
- **Xenomai:** It represents the third implementation. Its history is a bit twisted: it appeared in August 2001, only to be merged with RTAI in 2013 to generate a real-time operating system that was fit for production. However, the fusion was dispersed in 2005 and it became an independent project again.

The following diagram presents a basic RTLinux architecture.



A similar architecture, as shown in the preceding diagram, applies to the two other solutions since both of them were born from the RTLinux implementation. The difference between them is at the implementation level and each offers various benefits.

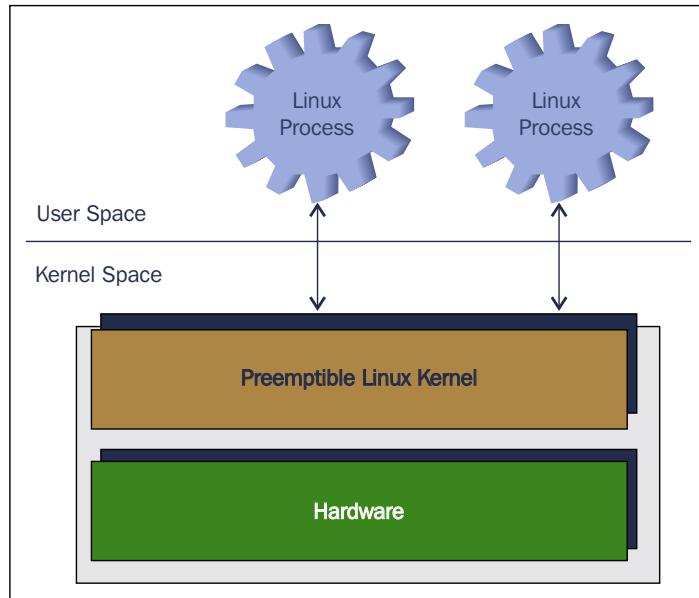
PREEMPT_RT

The PREEMPT_RT patches are the first option for every developer when a real-time solution is required. For some developers, the PREEMPT_RT patches transform Linux into a real-time solution suitable for their needs. This solution could not replace a real-time operation system, but is, in fact, suitable for a large number of systems.

The biggest advantage that PREEMPT_RT has over other real-time solutions for Linux is that it actually transforms Linux into a real-time operating system. All the other alternatives usually create a microkernel that is executed as a hypervisor and Linux is only executed as a task of it, so the communication of real-time tasks with the non-real-time ones is done through this microkernel. For the PREEMPT_RT patch, this problem is no more.

The standard version of the Linux kernel is only able to offer soft real-time requirements, such as basic POSIX user space operations where no deadline is guaranteed. Adding patches, such as Ingo Molnar's PREEMPT_RT patch, and also Thomas Gheixner's patch with regards to a generic clock event layer that offers a high resolution support, you can say that you have a Linux kernel that offers high real-time capabilities.

With the presence of the real-time preemption patch in the industry, a number of interesting opportunities have appeared, making it an option for firm and hard real-time applications in areas, such as industrial control or professional audio. This is mainly because of the design of the PREEMPT_RT patch and its aim toward integration inside the mainline kernel. We will learn about its usage further in the chapter. The following diagram shows the working of the Preemptible Linux Kernel:



The PREEMPT_RT patch transforms Linux from a general purpose operating system into a preemptible one using the following tricks:

- Protecting critical sections with the preemptible `rwlock_t` `preemptible` and `spinlock_t`. The use of the old solutions is still available using `raw_spinlock_t`, which shares the same API as `spinlock_t`.
- The kernel locking mechanisms is preempted by using `rtmutexes`.
- A priority inversion and priority inheritance mechanism is implemented for `mutexes`, `spinlocks` and `rw_semaphores`.
- Converting the available Linux timer API into one with a high resolution timer that offers the possibility of having timeouts.
- Implementing the usage of kernel threads for interrupt handlers. The real-time preemption patch treats soft interrupt handlers into the kernel thread context using a `task_struct` like structure for every user space process. There is also the possibility of registering an IRQ into the kernel context.



For more information on priority inversion, <http://www.embedded.com/electronics-blogs/beginner-s-corner/4023947/Introduction-to-Priority-Inversion> is a good starting point.

Applying the PREEMPT_RT patch

Before moving to the actual configuration part, you should download a suitable version for the kernel. The best inspiration source is <https://www.kernel.org/>, which should be the starting point because it does not contain any extra patches. After the source code is received, the corresponding rt patches version can be downloaded from <https://www.kernel.org/pub/linux/kernel/projects/rt/>. The kernel version chosen for this demonstration is the 3.12 kernel version, but if any other kernel version is required, the same steps can be taken with a similar end result. The development of the real-time preemption patches is very active, so any missing version support is covered very fast. Also, for other sublevel versions, the patches can be found in the `incr` or older subdirectories of that particular kernel version. The following is the example for sublevel versions:

```
wget https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.12.38.tar.xz
wget https://www.kernel.org/pub/linux/kernel/projects/rt/3.12/patch-3.12.38-rt52.patch.gz
```

After the source code is received, the sources need to be unpacked and the patches applied:

```
tar xf linux-3.12.38.tar.xz
cd linux-3.12.38/
gzip -cd ../patch-3.12.38-rt52.patch.gz | patch -p1
```

The next step involves the configuration of the kernel sources. The configuration differs from one architecture to another, but the general idea remains. The following configurations are required for a QEMU ARM machine supported inside Poky. To enable the PREEMPT_RT support for a machine, there are multiple options available. You can implement a low-latency support version, which is most suitable for a desktop computer using a kernel configuration fragment similar to this:

```
CONFIG_GENERIC_LOCKBREAK=y
CONFIG_TREE_PREEMPT_RCU=y
CONFIG_PREEMPT_RCU=y
CONFIG_UNINLINE_SPIN_UNLOCK=y
CONFIG_PREEMPT=y
```

```
CONFIG_PREEMPT__LL=y
CONFIG_PREEMPT_COUNT=y
CONFIG_DEBUG_PREEMPT=y
CONFIG_RCU_CPU_STALL_VERBOSE=y
```

This option is one of the most often used and it also constitutes the primary source of usage of the PREEMPT_RT patches. The alternative of this would be to enable the fully preemptive support for the PREEMPT_RT patches using a configuration similar to this:

```
CONFIG_PREEMPT_RT_FULL=y
CONFIG_HZ_1000=y
CONFIG_HZ=1000
```

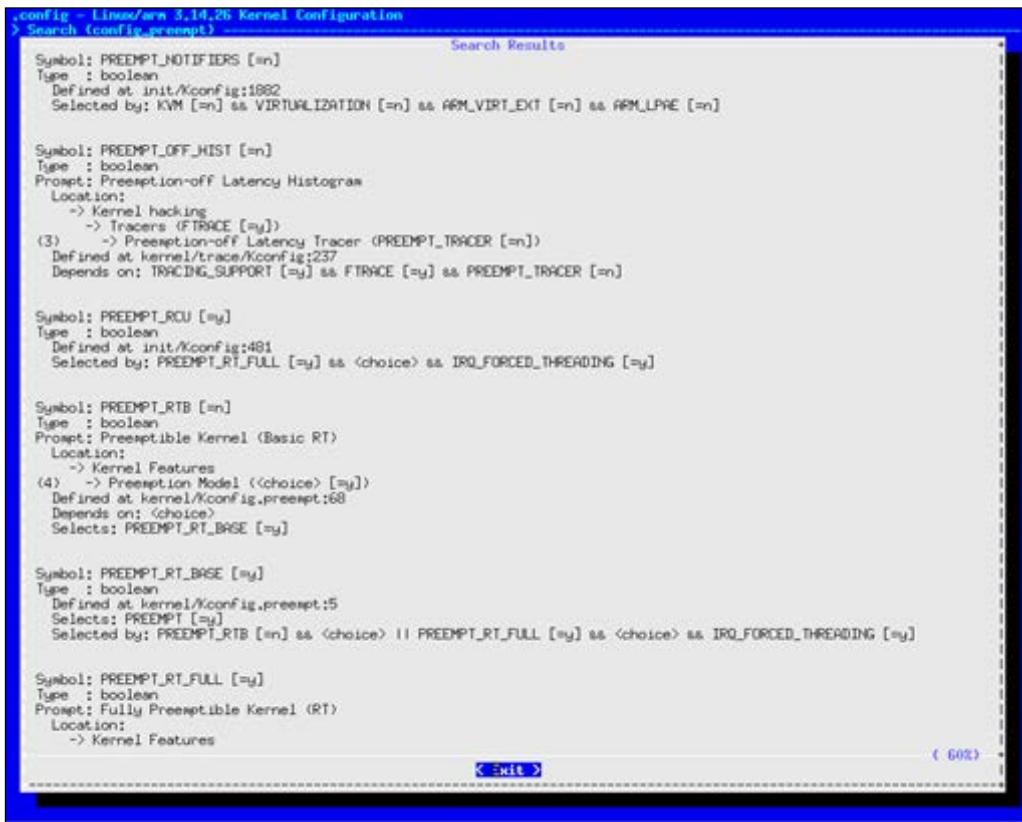
If you're interested in configuring the kernel manually, it can use the menuconfig option. The following CONFIG_PREEMPT* configurations are available for easier access to the required options. The first image mainly contains the CONFIG_PREEMPT and CONFIG_PREEMPT_COUNT variables, which should be the first ones to enable. There is also a configuration option called CONFIG_PREEMPT_NONE that is used for no forced preemptive actions.

The screenshot shows a terminal window titled "config - Linux/arm 3.14.26 Kernel Configuration". It displays the search results for the symbol "PREEMPT". The results list several kernel configuration symbols related to preemptiveness:

- Symbol: PREEMPT [=y]
Type : boolean
Defined at kernel/Kconfig.preempt:1
Selects: PREEMPT_COUNT [=y]
Selected by: PREEMPT_RT_BASE [=y] || PREEMPT__LL [=n] as <choice>
- Symbol: DEBUG_PREEMPT [=y]
Type : boolean
Prompt: Debug preemptible kernel
Location:
 - (1) -> Kernel hacking
Defined at lib/Kconfig.debug:808
Depends on: DEBUG_KERNEL [=y] && PREEMPT [=y] && TRACE_IROFLAGS_SUPPORT [=y]
- Symbol: HAVE_PREEMPT_LAZY [=y]
Type : boolean
Defined at kernel/Kconfig.preempt:9
Selected by: ARM [=y]
- Symbol: PREEMPT_COUNT [=y]
Type : boolean
Defined at kernel/Kconfig.preempt:86
Selected by: PREEMPT [=y] || DEBUG_ATOMIC_SLEEP [=n] && DEBUG_KERNEL [=y]
- Symbol: PREEMPT_LAZY [=y]
Type : boolean
Defined at kernel/Kconfig.preempt:12
- Symbol: PREEMPT_NONE [=n]
Type : boolean
Prompt: No Forced Preemption (Server)
Location:
 - > Kernel Features
 - (2) -> Preemption Model (<choice> [=y])
Defined at kernel/Kconfig.preempt:19
Depends on: <choice>
- Symbol: PREEMPT_NOTIFIERS [=n]
Type : boolean
Defined at init/Kconfig:1882

The terminal window has a blue border and a black background. The text is white or light gray. At the bottom right, there is a status bar with the number "(28)".

In the following image, the CONFIG_PREEMPT_RCU and CONFIG_PREEMPT_RT_FULL configurations are available. More information related to RCU is available at <https://lwn.net/Articles/262464/>.



The screenshot shows the 'Search (config preempt)' results in the 'config - Linux/zen 3.14.26 Kernel Configuration' window. The results list several configuration symbols:

- Symbol: PREEMPT_NOTIFIERS [=n]**
Type : boolean
Defined at init/Kconfig:1882
Selected by: KVM [=n] & VIRTUALIZATION [=n] & ARM_VIRT_EXT [=n] & ARM_LPAE [=n]
- Symbol: PREEMPT_OFF_HIST [=n]**
Type : boolean
Prompt: Preemption-off Latency Histogram
Location:
 - > Kernel hacking
 - > Tracers (FTRACE [=y])(3) -> Preemption-off Latency Tracer (PREEMPT_TRACER [=n])
Defined at kernel/trace/Kconfig:237
Depends on: TRACING_SUPPORT [=y] & FTRACE [=y] & PREEMPT_TRACER [=n]
- Symbol: PREEMPT_RCU [=y]**
Type : boolean
Defined at init/Kconfig:481
Selected by: PREEMPT_RT_FULL [=y] & <choice> & IRQ_FORCED_THREADING [=y]
- Symbol: PREEMPT_RT_B [in]**
Type : boolean
Prompt: Preemptible Kernel (Basic RT)
Location:
 - > Kernel Features(4) -> Preemption Model (<choice> [=y])
Defined at kernel/Kconfig.preempt:68
Depends on: <choice>
Selects: PREEMPT_RT_BASE [=y]
- Symbol: PREEMPT_RT_BASE [=y]**
Type : boolean
Defined at kernel/Kconfig.preempt:5
Selects: PREEMPT [=y]
Selected by: PREEMPT_RT_B [=n] & <choice> || PREEMPT_RT_FULL [=y] & <choice> & IRQ_FORCED_THREADING [=y]
- Symbol: PREEMPT_RT_FULL [=y]**
Type : boolean
Prompt: Fully Preemptible Kernel (RT)
Location:
 - > Kernel Features

The status bar at the bottom right shows '(60%)'.

The third image contains the CONFIG_PREEMPT_LL configuration. Another interesting configuration is CONFIG_PREEMPT_VOLUNTARY, which also reduces the latency along with the CONFIG_PREEMPT_LL configuration, for a desktop computer.

Real-time

One interesting argument against the *low-latency desktop* option is available at <https://sevencapitalsins.wordpress.com/2007/08/10/low-latency-kernel-wtf/>.

The screenshot shows the Linux kernel configuration interface for a 3.14.26 build. A search has been performed for the word "preempt". The results list several kernel configuration symbols:

- Symbol: PREEMPT_RT_BASE [=y]**
Type : boolean
Defined at kernel/Kconfig.preempt:5
Selects: PREEMPT [=y]
Selected by: PREEMPT_RT [=n] as <choice>; PREEMPT_RT_FULL [=y] as <choice> & IRQ_FORCED_THREADING [=y]
- Symbol: PREEMPT_RT_FULL [=y]**
Type : boolean
Prompt: Fully Preemptible Kernel (RT)
Location:
 - > Kernel Features(5) -> Preemption Model (<choice> [=y])
Defined at kernel/Kconfig.preempt:76
Depends on: <choice> & IRQ_FORCED_THREADING [=y]
Selects: PREEMPT_RT_BASE [=y] as PREEMPT_RCU [=y]
- Symbol: PREEMPT_TRACER [=n]**
Type : boolean
Prompt: Preemption-off Latency Tracer
Location:
 - > Kernel hacking(6) -> Tracers (FTRACE [=y])
Defined at kernel/trace/kconfig:213
Depends on: TRACING_SUPPORT [=y] as FTRACE [=y] & IARCH_USES_GETTIMEOFFSET [=n] & PREEMPT [=y]
Selects: GENERIC_TRACER [=y] as TRACER_MAX_TRACE [=n] as RING_BUFFER_ALLOW_SWAP [=n] as TRACER_SNAPSHOT [=n] as TRACER
- Symbol: PREEMPT_VOLUNTARY [=n]**
Type : boolean
Prompt: Voluntary Kernel Preemption (Desktop)
Location:
 - > Kernel Features(7) -> Preemption Model (<choice> [=y])
Defined at kernel/Kconfig.preempt:32
Depends on: <choice>
- Symbol: PREEMPT_LL [=n]**
Type : boolean
Prompt: Preemptible Kernel (Low-Latency Desktop)
Location:
 - > Kernel Features(8) -> Preemption Model (<choice> [=y])
Defined at kernel/Kconfig.preempt:49

The search results window has a title bar "Search (Config.preempt)" and a status bar at the bottom right showing "(89%)".

The last one contains the CONFIG_TREE_PREEMPT_RCU configuration used to change the RCU implementation. The same process can be used to search and enable the other configurations that do not contain the search word in their name.

The screenshot shows the 'Search Results' window of the Linux kernel configuration tool. The search term is 'preempt'. The results list several kernel symbols and their configurations:

- Symbol: PREEMPT_TRACER [=n]**
Type : boolean
Prompt: Preemption-off Latency Tracer
Location:
 -> Kernel Tracers (FTRACE [=y])
 Defined at kernel/trace/Kconfig:213
 Depends on: TRACING_SUPPORT [=y] & FTRACE [=y] & ARCH_USES_GETTIMEOFFSET [=n] & PREEMPT [=y]
 Selects: GENERIC_TRACER [=y] & TRACER_MAX_TRACE [=n] & RING_BUFFER_ALLOW_SWP [=n] & TRACER_SNAPSHOT [=n] & TRACER_CONTEXT [=n]
- Symbol: PREEMPT_VOLUNTARY [=n]**
Type : boolean
Prompt: Voluntary Kernel Preemption (Desktop)
Location:
 -> Kernel Features
 (7) -> Preemption Model (<choice> [=y])
 Defined at kernel/Kconfig.preempt:32
 Depends on: <choice>
- Symbol: PREEMPT_LL [=n]**
Type : boolean
Prompt: Preemptible Kernel (Low-Latency Desktop)
Location:
 -> Kernel Features
 (8) -> Preemption Model (<choice> [=y])
 Defined at kernel/Kconfig.preempt:49
 Depends on: <choice>
 Selects: PREEMPT [=y] & UNINLINE_SPIN_UNLOCK [=n]
- Symbol: TREE_PREEMPT_RCU [=y]**
Type : boolean
Prompt: Preemptible tree-based hierarchical RCU
Location:
 -> General setup
 -> RCU Subsystems
 (9) -> RCU Implementation (<choice> [=y])
 Defined at init/Kconfig:457
 Depends on: <choice> & PREEMPT [=y]
 Selects: IRQ_WORK [=y]

At the bottom right of the window, there is a progress bar indicating '(100%)' and a button labeled '< Exit >'.

For more information regarding the PREEMPT_RT patch, refer to <http://varun-anand.com/preempt.html> and http://www.versalogic.com/mediacenter/whitepapers/wp_linux_rt.asp.

After the kernel image is obtained with the newly applied and configured real-time preemptible kernel patch, it needs to be booted to make sure the activity is done appropriately so that the end result can be usable. Using the `uname -a` command, the patch `rt*` revision number is visible and should be applied to the kernel version. Of course, there are other methods that can be used to identify this information. An alternative for the `uname -a` command is the `dmesg` command on its output the string real-time preemption support should be visible, but only one method should be enough. The following image offers a representation of how the `uname -a` command output should look:

```
root@qemuarm: # uname -a
Linux qemuarm 3.14.19-rt9-yocto-preempt-rt #1 PREEMPT RT Mon Mar 9 15:16:22 CET
2015 armv5tejl GNU/Linux
```

Taking a look at the list of processes, it can be seen, as mentioned earlier, that the IRQ handler is treated using kernel threads. This information is visible in the next `ps` command output due to the fact that it is put between square brackets. Single IRQ handlers are represented by the `task_struct` structures that are similar to the user space ones, making them easily controllable from the user space:

```
ps ax
PID TTY      STAT   TIME COMMAND
1 ?        S      0:00 init [2]
2 ?        S      0:00 [softirq-high/0]
3 ?        S      0:00 [softirq-timer/0]
4 ?        S      0:00 [softirq-net-tx/]
5 ?        S      0:00 [softirq-net-rx/]
6 ?        S      0:00 [softirq-block/0]
7 ?        S      0:00 [softirq-tasklet]
8 ?        S      0:00 [softirq-hrtreal]
9 ?        S      0:00 [softirq-hrtmono]
10 ?       S<    0:00 [desched/0]
11 ?       S<    0:00 [events/0]
12 ?       S<    0:00 [khelper]
13 ?       S<    0:00 [kthread]
15 ?       S<    0:00 [kblockd/0]
58 ?       S      0:00 [pdflush]
59 ?       S      0:00 [pdflush]
61 ?       S<    0:00 [aio/0]
60 ?       S      0:00 [kswapd0]
647 ?      S<    0:00 [IRQ 7]
648 ?      S<    0:00 [kseriod]
651 ?      S<    0:00 [IRQ 12]
654 ?      S<    0:00 [IRQ 6]
675 ?      S<    0:09 [IRQ 14]
687 ?      S<    0:00 [kpsmoused]
689 ?      S      0:00 [kjournald]
691 ?      S<    0:00 [IRQ 1]
769 ?      S<s   0:00 udevd --daemon
871 ?      S<    0:00 [khubd]
```

```
882 ?      S<      0:00 [IRQ 10]
2433 ?      S<      0:00 [IRQ 11]
[...]
```

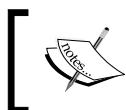
The next bit of information that needs to be gathered involves the formatting of the interrupt process entries, which are a bit different than the ones used for a vanilla kernel. This output is visible by inspecting the `/proc/interrupts` file:

```
cat /proc/interrupts
CPU0
0:      497464 XT-PIC      [.....N/ 0]  pit
2:          0 XT-PIC      [.....N/ 0]  cascade
7:          0 XT-PIC      [.....N/ 0]  lpptest
10:         0 XT-PIC      [...../ 0]  uhci_hcd:usb1
11:        12069 XT-PIC      [...../ 0]  eth0
14:        4754 XT-PIC      [...../ 0]  ide0
NMI:        0
LOC:       1701
ERR:        0
MIS:        0
```

Then, information available in the fourth column provides the IRQ line notifications, such as: `[.....N/ 0]`. Here, each dot represents an attribute and each attribute is a value, as described in the following points. Here is the order of their presence:

- I (IRQ_INPROGRESS): This refers to the IRQ handler that is active
- D (IRQ_DISABLED): This represents the IRQ as being disabled
- P (IRQ_PENDING): The IRQ here is presented as being in a pending state
- R (IRQ_REPLY): In this state, the IRQ has been replied to, but no ACK is received yet
- A (IRQ_AUTODETECT): This represents the IRQ as being in an autodetect state
- W (IRQ_WAITING): This refers to the IRQ being in an autodetect state, but not seen yet
- L (IRQ_LEVEL): The IRQ is in a level-triggered state
- M (IRQ_MASKED): This represents the state in which the IRQ is not visible as being masked anymore
- N (IRQ_NODELAY): This is the state in which the IRQ must be executed immediately

In the preceding example, you can see that multiple IRQs are marked as visible and hard IRQs that are run in the kernel context. When an IRQ status is marked as `IRQ_NODELAY`, it shows the user that the handler of the IRQ is a kernel thread and it will be executed as one. The description of an IRQ can be changed manually, but this is not an activity that will be described here.



For more information on how to change the real-time attributes for a process, a good starting point is the `chrt` tool, available at <http://linux.die.net/man/1/chrt>.



The Yocto Project -rt kernel

Inside Yocto, kernel recipes with `PREEMPT_RT` patches are applied. For the moment, there are only two recipes that incorporate the `PREEMPT_RT` patch; both are available inside the meta layer. The recipes that refer to kernel versions 3.10 and 3.14 and their naming are `linux-yocto-rt_3.10.bb` and `linux-yocto-rt_3.14.bb`. The `-rt` ending in the naming indicates that these recipes fetch the `PREEMPT_RT` branches of the Linux kernel versions maintained by the Yocto community.

The format for the 3.14 kernel recipe is presented here:

```
cat ./meta/recipes-kernel/linux/linux-yocto-rt_3.14.bb
KBRANCH ?= "standard/preempt-rt/base"
KBRANCH_qemuppc ?= "standard/preempt-rt/qemuppc"

require recipes-kernel/linux/linux-yocto.inc

SRCREV_machine ?= "0a875ce52aa7a42ddabdb87038074381bb268e77"
SRCREV_machine_qemuppc ?=
"b993661d41f08846daa28b14f89c8ae3e94225bd"
SRCREV_meta ?= "fb6271a942b57bdc40c6e49f0203be153699f81c"

SRC_URI = "git://git.yoctoproject.org/linux-yocto-3.14.git;
bareclone=1;branch=${KBRANCH},meta;name=machine,meta"

LINUX_VERSION ?= "3.14.19"

PV = "${LINUX_VERSION}+git${SRCPV}"

KMETA = "meta"
```

```
LINUX_KERNEL_TYPE = "preempt-rt"

COMPATIBLE_MACHINE = "(qemux86|qemux86-
64|qemuarm|qemuppc|qemumips)"

# Functionality flags
KERNEL_EXTRA_FEATURES ?= "features/netfilter/netfilter.scc
features/taskstats/taskstats.scc"
KERNEL_FEATURES_append = "${KERNEL_EXTRA_FEATURES}"
KERNEL_FEATURES_append_qemux86=" cfg/sound.scc
cfg/paravirt_kvm.scc"
KERNEL_FEATURES_append_qemux86=" cfg/sound.scc
cfg/paravirt_kvm.scc"
KERNEL_FEATURES_append_qemux86-64=" cfg/sound.scc"
```

As shown, one of the recipes seemed to have a duplicated line and a patch is necessary to remove it:

```
commit e799588ba389ad3f319afdf1a61e14c43fb78a845
Author: Alexandru.Vaduva <Alexandru.Vaduva@enea.com>
Date:   Wed Mar 11 10:47:00 2015 +0100

    linux-yocto-rt: removed duplicated line

    Seemed that the recipe contained redundant information.

    Signed-off-by: Alexandru.Vaduva <Alexandru.Vaduva@enea.com>

diff --git a/meta/recipes-kernel/linux/linux-yocto-rt_3.14.bb
b/meta/recipes-kernel/linux/linux-yocto-rt_3.14.bb
index 7dbf82c..bcfd754 100644
--- a/meta/recipes-kernel/linux/linux-yocto-rt_3.14.bb
+++ b/meta/recipes-kernel/linux/linux-yocto-rt_3.14.bb
@@ -23,5 +23,4 @@ COMPATIBLE_MACHINE = "(qemux86|qemux86-
64|qemuarm|qemuppc|qemumips)"
KERNEL_EXTRA_FEATURES ?= "features/netfilter/netfilter.scc
features/taskstats/taskstats.scc"
KERNEL_FEATURES_append = "${KERNEL_EXTRA_FEATURES}"
KERNEL_FEATURES_append_qemux86=" cfg/sound.scc
cfg/paravirt_kvm.scc"
-KERNEL_FEATURES_append_qemux86=" cfg/sound.scc
cfg/paravirt_kvm.scc"
KERNEL_FEATURES_append_qemux86-64=" cfg/sound.scc"
```

The preceding recipe is very similar to the base one. Here, I am referring to `linux-yocto_3.14.bb`; they are the recipes on which the PREEMPT_RT patches have been applied. The difference between them is that each one is taken from its specific branch, and until now, none of the Linux kernel versions with the PREEMPT_RT patches have provided support for the `qemumips64` compatible machine.

Disadvantages of the PREEMPT_RT patches

Linux, a general purpose operating system that is optimized for throughput, is the exact opposite of what a real-time operating system is all about. Of course it offers a high throughput by using a large, multilayered cache, which is a nightmare for a hard real-time operating process.

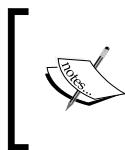
In order to have a real-time Linux, there are two available options:

- The first one involves the use of the PREEMPT_RT patches, which offer preemption by minimizing the latency and executing all activities in a thread context.
- The second solution involves the use of real-time extensions that act as layers between Linux and the hardware used for the management of real-time tasks. This second solution includes the previously mentioned RTLinux, RTAI, and XENOMAI solutions, as well as other commercial solutions and variations that involve moving the layer and also separating it in multiple components.

The variations of the second option imply various solution from the isolation of the cores for real-time activities to the assignation of one for such tasks. There are also a lot of solutions that involve the usage of a hypervisor or a hook below the Linux kernel to serve a number of interrupts to the RTOS. The existence of these alternatives have been made available to the reader not only with other options, but also due to the fact that the PREEMPT_RT patch has its disadvantages.

One notable disadvantage is that the reduction of latency was done by forcing the kernel to preempt a task when a higher priority one appeared. This, of course, reduces the throughput for the system because it not only adds a number of context switches in the process but also makes the lower priority tasks wait longer than they would do the normal Linux kernel.

Another disadvantage of the `preempt-rt` patches is that they need to be ported from one kernel version to another and adapted from one architecture or software vendor to another. This only implies that knowledge of the Linux kernel should be available in-house for a particular vendor and it should adapt the solution for each of its available kernels. This fact alone has made it less likeable for BSP or Linux operating system providers.



One interesting presentation regarding the Linux pre-emption is available in the following link. It can be consulted for more information regarding a Linux real-time solution, and is available at <http://www.slideshare.net/jserv/realtime-linux>.

Linux real-time applications

Having a real-time operating system may not always be enough for everyone. Some people would also require real-time optimized applications running over the operating system. To make sure an rt-application can be designed and interacted with, the required determinism is necessary on the operating system and hardware. With regard to the hardware configuration, the requirements involve a low-latency interrupt handling. The mechanisms causing the ISR latencies should register values around tens of microseconds.

Regarding the kernel configuration required by real-time applications, the following configurations are necessary:

- **On-demand CPU scaling:** Using this configuration helps with the creation of long-latency events when the CPU is in a low-power consumption mode.
- **NOHZ:** This configuration disables the timer interrupt received by CPUs. With this option enabled, the latency spent on a CPU wake up is diminished.

To write an application, there are some things that need to be taken care of, such as making sure that the use of swap is disabled to diminish latencies caused by page faults. The use of global variables or arrays should be kept to a minimum. The 99 priority number is not configured to run an application, and other spin locks are not implemented instead, it uses priority inheritance futexes. Also avoid input/output operations and data sharing between applications.

For a device driver, the advice is a bit different. Previously, we mentioned that the interrupt handling for a real-time kernel is done in a thread context, but the hardware interrupt context can still play a role here. To recognize the hardware interrupt context from the interrupt handler, the `IRQF_NODELAY` flag can be used. If you use the `IRQF_NODELAY` context, make sure you avoid functions such as `wake_up()`, `up()`, or `complete()`.

Benchmarking

The Linux operating system was for a very long time seen as a GPOS, but in the last couple of years, some projects tried to change this by modifying the Linux kernel into a RTOS. One such project is the PREEMPT_RT patch, which was mentioned previously.

In this section of the chapter, I will discuss a series of tests that could be executed for both versions of the Linux OS with or without applying the PREEMPT_RT patches. I should mention that for those of you who are interested in some actual results, there are a number of papers available that try to investigate the latency effect of the PREEMPT_RT or its advantages or disadvantages. One such example is available at http://www.versalogic.com/downloads/whitepapers/real-time_linux_benchmark.pdf.

Before continuing further, I believe it is my duty to define a number of technical terms that are necessary to properly understand some information:

- **Interrupt latency:** This indicates the time that has elapsed since an interrupt was generated and until the execution has been started in the interrupt handler.
- **Scheduling latency:** This represents the time between the wake up signal of an event and a scheduler that has the opportunity to schedule a thread for it. It is also called a **dispatch latency**.
- **Worst-case latency:** This indicates the time that has passed since a demand was issued and until the response to that demand was received.
- **Context-switch:** This represents the switching of the CPU from one process or thread to another. It only occurs in the kernel mode.

The **LPPTest** is included in the PREEMPT_RT patch and it contains a Linux driver that only changes a bit value on a parallel port to identify the response time. Another driver responds to the change in a bit value and a user space application that measures the results. The files to look for are `drivers/char/lpptest.c` and `scripts/testlpp.c`. To perform this test, two machines are required: one to send the signal and the other one to receive and send the response. This requirement is stringent since the use of a loopback cable can mess with the measurements.

RealFeel is a test for interrupt processing. The program uses `/dev/rtc` to fire a periodic interrupt, measures the duration between one interrupt to another, and compares it with the expected value. At the end, it prints the variation from the expected value indefinitely so that the variations can be exported in a log file to process later.

Linux Real-Time Benchmarking Framework (LRTB) represents a set of scripts and drivers that are used to evaluate various performance counters for the Linux kernel with a real-time addition. It measures the load imposed by real-time patches and their ability to obtain a more deterministic response to interrupts.

For the benchmarking phase, programs such as `hackbench`, `lmbench`, or even the Ingo Molnar `dohell` script can be used. There are, of course, a number of other tools that can be used for both testing (`cyclictest`, `hourglass`, and so on) or benchmarking (`unixbench`, `cache-calibrator`, or any other stress test that takes real-time performances to their limit), but I will let the user test them and apply the ones that suit their needs best.

The PREEMPT_RT patch improves the preemptiveness of the Linux kernel, but this does not mean it is the best solution to use. The usefulness of PREEMPT_RT patch can differ if various aspects of the application domain changes. With regard to the PREEMPT_RT patch, it is ready to be used in a hard real-time system. One conclusion cannot be made, but I must admit that it can be considered hard real-time material if it is used in life sustaining or mission-critical systems. This is a decision for everybody to make, and for this testing is required. One opinion that supports this is from Steven Rostedt, a Linux kernel developer who is the maintainer of the stable version of the real-time Linux kernel patch for Red Hat. It is available at <http://www.linux.com/news/featured-blogs/200-libby-clark/710319-intro-to-real-time-linux-for-embedded-developers>.



Some interesting information on this matter can be accessed at
http://elinux.org/Realtime_Testing_Best_Practices.



Meta-realtime

The meta-realtime layer is an initiative maintained by Bruce Ashfield from WindRiver, which planned to create a place where real-time activities related to the Linux kernel or system development. It was created as the placeholder for PREEMPT_RT, SCHED_DEADLINE, POSIX real-time, and alternative paring of general purpose operating systems and real-time operating systems, whether this involved a user space RTOS, a hypervisor, or an AMP solution. Also, this is where system partitioning, CPU isolation, and other related applications s reside. Of course, none of this would be considered complete without some performance profiling and benchmarking applications available for the whole Linux operating system.

Although this layer description sounds really exciting at first, its content is really poor. It is only able to incorporate a number of testing tools, more accurately, two of them: `schedtool-dl` and `rt-app`, as well as extra scripts that try to remotely run `rt-app` on the target machine and gather the resulting data.

The first `schedtool-dl` application is a scheduler testing tool used for deadline scheduling. It appears from the need to change or make queries of the CPU-scheduling policies and even processes levels available under Linux. It can also be used to lock processes on various CPUs for SMP/NUMA systems, to avoid skipping in audio/video applications, and in general, to maintain a high level of interaction and responsiveness even under high loads.



More information about the `schedtool-dl` application can be found at <https://github.com/jlelli/schedtool-dl>.



The next and last available application is `rt-app`, which is used as a test application for the simulation of real-time loads on a system. It does this by starting multiple threads at given periods of time. It offers support for `SCHED_FIFO`, `SCHED_OTHER`, `SCHED_RR`, `SCHED_DEADLINE`, as well as the **Adaptive Quality of Service Architecture (AQuoSA)** framework, which is an open source project that tries to offer adaptive **Quality of Service (QoS)** for the Linux kernel.



More information about the `rt-app` application and the AQuoSa framework can be found at <https://github.com/scheduler-tools/rt-app> and <http://aquosa.sourceforge.net/>.



Besides the included packages, the layer also contains an image that incorporates them, but this is not nearly enough to make this layer one that contains substantial content. Although it does not contain a vast amount of information inside it, this layer has been presented in this chapter because it contains the starting point and offers a development point of view of all the information presented until now. Of course, a number of applications that should reside in this layer are already spread across multiple other layers, such as the `idlestat` package that is available in `meta-linaro`. However, this does not constitute the central point of this explanation. I only wanted to point out the most suitable place that can contain any real-time relate activities, and in my opinion, `meta-realtime` is this place.

Summary

In this chapter, you were given a short introduction to PREEMPT_RT and other alternative solutions for real-time problems of the Linux kernel. We also explored a number of tools and applications that can be used for related real-time activities. However, this presentation would not be complete without references made to the Yocto Project with regards not only to the recipes of the PREEMPT_RT Linux kernel, but also to `meta-realtime` layer applications. Developing an application suitable for a new context was also a concern, so this problem was tackled in the *Linux real-time applications* section. In the end, I hope that I was able to present a complete picture of this subject through links that were provided throughout the chapter to stir the curiosity of the reader.

In the next chapter, a short explanation of `meta-security` and `meta-selinux` layers will be given and a broader picture of the security requirements of the Linux ecosystem in general and the Yocto Project in particular, will be provided. Information regarding a number of tools and applications that try to secure our Linux systems will also be presented, but this is not all. Take a look at the next chapter; I am sure you will enjoy it.

11

Security

In this chapter, you will be presented with various security enhancements tools. Our first stop is the Linux kernel and here, there are two tools, SELinux and grsecurity, both of which are really interesting as well as necessary. Next, the Yocto Project's security-specific layers will also be explained. These include the meta-security and meta-selinux that contain an impressive number of tools and can be used to secure or audit various components of the Linux system. Since this subject is vast, I will also let you inspect various other solutions, both implemented in the Linux kernel but also externally. I hope you enjoy this chapter and that you find this information interesting and useful.

In any operating system, security is a really important concern both for the users and developers. It did not pass much time and developers have started to address these security problems in various methods. This resulted in a number of security methodologies and improvements for available operating systems. In this chapter, a number of security enhancement tools will be introduced along with some policies and verification routines that are defined to ensure that various components, such as the Linux kernel or the Yocto Project, are secure enough to be used. We will also take a look at how various threats or problems are handled as they appear during the course of this chapter.

SELinux and grsecurity are two noticeable security improvements made to the Linux kernel that try to enforce Linux. SELinux is a **Mandatory Access Control (MAC)** mechanism that provides identity and role-based access control as well as domain-type enforcement. The second option, grsecurity, is more similar to ACLs and is, in fact, more suitable for web servers and other systems that support remote connections. With regard to how security is implemented for Linux and how the Yocto Project handles this domain, these aspects will be presented in the next section. One thing I must admit is that security handling inside the Yocto Project is still a young project at the time of writing this chapter, but I am waiting with enthusiasm to see how the number of iterations will increase over time.

Security in Linux

At the core of every Linux system is the Linux kernel. Any malicious code that is able to damage or take control of a system also has repercussions that affect the Linux kernel. So, it only makes clear to users that having a secure kernel is also an important part of the equation. Fortunately, the Linux kernel is secure and has a number of security features and programs. The man behind all this is James Morris, the maintainer of the Linux kernel security subsystem. There is even a separate Linux repository for this that can be accessed at <http://git.kernel.org/?p=linux/kernel/git/jmorris/linux-security.git;a=summary>. Also, by inspecting http://kernsec.org/wiki/index.php/Main_Page, which is the main page of the Linux kernel security subsystem, you can see the exact projects that are managed inside this subsystem and maybe lend a hand to them if you're interested.

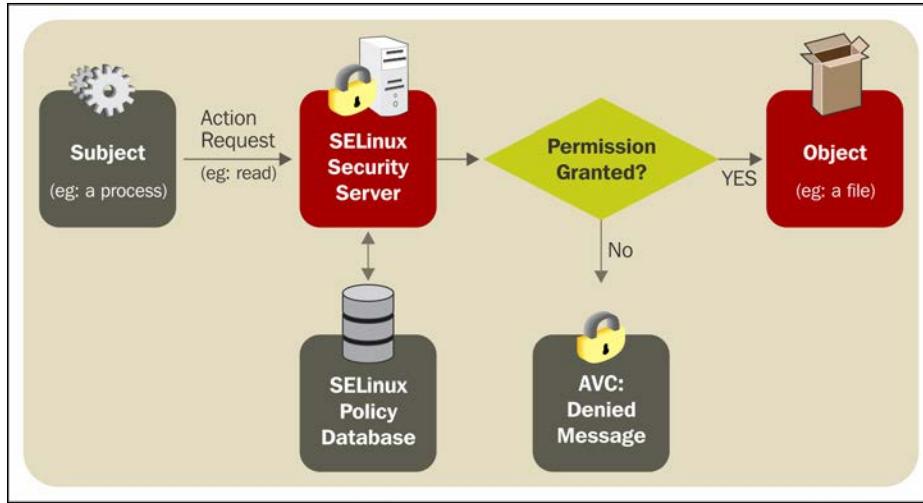
There is also a workgroup that provides security enhancements and verifications to the Linux kernel to make sure that it is secure and also to maintain a certain level of trust in the security of the Linux ecosystem. Their activities include, but of course, are not limited to verification and testing of critical subsystems for various vulnerabilities or the development of tools to assist in the security Linux kernel. The workgroup also consists of guidance and maintenance of security subsystems or security improvements added to various projects or build tools.

All the other Linux software components have their own security teams. Of course, there are some that do not have these teams well defined, or have some internal rules related to this subject, but they are still aware of security threats that occur around their components and try to repair these vulnerabilities. The Yocto Project tries to help with these problems and in some ways unifies these software components. I hope that some improvements are made over the years in this area.

SELinux

SELinux is a security enhancement for the Linux kernel, and is developed by the National Security Agency's office of Information Assurance. It has a policy-based architecture and is one of the Linux security modules that is built on the interface of **Linux Security Modules (LSM)** that aims at military-level security.

Currently, it is shipped with a large number of distributions, including the most well known and often used ones, such as Debian, SuSe, Fedora, Red Hat, and Gentoo. It is based on MAC on which administrators can control all interactions with the user space components of a system. It uses the concept of least privileges: here, by default, a user and application have no rights to access the system resources since all of them are granted by an administrator entity. This makes up the part of the system security policies and its emphasis is shown in the following figure:



The basic functionalities inside SELinux are sandboxed with the help of the implementation of MAC. Inside the sandbox, each application is allowed to perform only the task it was designed to execute as defined in the security policies. Of course, standard Linux permissions are still available for the system and they will be consulted before the policies when access attempts are required. If no permissions are available, SELinux will not be able to influence the system in any way. However, if the permission rights allow access, then the SELinux policies should be consulted to offer the final verdict on whether access is permitted or denied.

In the context of SELinux, access decisions are made based on the security context of the subject. This may very well be a process associated with a specific user context that is compared with the actual attempted action (such as a file read action), and the security context of the available object, which can be a file.

Before moving on, we will see how the SELinux support can be enabled on a Ubuntu machine. I will first present some basic concepts related to SELinux:

- **Users:** In the SELinux context, the user is not the same as the one available in the UNIX context. The major difference between them is that, in the SELinux context, the user does not change during a user session and there is a possibility for more UNIX users to operate in the same SELinux user context. However, there is also a possibility of operating in a 1:1 user mapping, such as the Linux root user and the SELinux root user. Generally, the SELinux users have the `_u` suffix added to their naming.
- **Roles:** A SELinux user can have one or multiple roles. The meaning of a role is defined in the policies. An object usually has the `object_r` role and the role is generally suffixed with the `_r` string.

- **Types:** It's the primary method applied to take authorization decisions. It can also be referred to as a domain and is generally suffixed with `_t`.
- **Contexts:** Each process and object has its context. It is, in fact, an attribute that determines whether access should be allowed between an object and process. A SELinux context is expressed as three required fields and as an optional one as well, such as `user:role:type:range`. The first three fields represent the SELinux user, role, and the type. The last one represents the range of MLS and it will be presented shortly. More information about MLS can be gathered at http://web.mit.edu/rhel-doc/5/RHEL-5-manual/Deployment_Guide-en-US/sec-mls-ov.html.
- **Object Classes:** An SELinux object class represents the category of objects available. Categories, such as `dir` for directories and `file` for files, also have a set of permissions associated with them.
- **Rules:** These are the security mechanisms of SELinux. They are used as a type of enforcement and are specified using the type of the object and process. The rules usually state if a type is allowed to perform various actions.

As mentioned already, the SELinux is so well known and appreciated that it was included in most of the available Linux distributions. Its success is also demonstrated through the fact that a huge number of books were written on this subject. For more information regarding it, refer to http://www.amazon.com/s/ref=nb_ss_gw/102-2417346-0244921?url=search-alias%3Daps&field-keywords=SELinux&Go.x=12&Go.y=8&Go=Go. Having said this, let's take a look at the steps required to install SELinux on a Ubuntu host machine. The first step refers to the SELinux package installation:

```
sudo apt-get install selinux
```

With the package installed, the SELinux mode needs to be changed from disabled (the mode in which the SELinux policy is not enforced or logged) to one of the other two available options:

- **Enforcing:** This is most useful in a production system:

```
sudo sed -i 's/SELINUX=.*/SELINUX=enforcing/' /etc/selinux/config
```
- **Permissive:** In this mode, policies are not enforced. However, any denials are logged and it is mostly used in debugging activities and when new policies are developed:

```
sudo sed -i 's/SELINUX=.*/SELINUX=permissive/' /etc/selinux/config
```

With the configuration implemented, the system needs to reboot, to make sure that the system files are labeled accordingly.

More information about SELinux is also available in the Yocto Project. There is an entire layer dedicated to SELinux support. Also, for more information regarding this tool, you are encouraged to read one of the books dedicated to this matter. If you dislike this method, then there are alternative manuals with information related to SELinux, available inside various distributions, such as Fedora (https://docs.fedoraproject.org/en-US/Fedora/19/html/Security_Guide/ch09.html), Red Hat (https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/4/html/SELinux_Guide/index.html), and so on.

Grsecurity

Grsecurity is a suite of patches released under the GNU General Public License, available for the Linux kernel and will help with the security enhancements for Linux. This suite of patches offers four main benefits:

- Configuration-free operations
- Protection against a large variety of address space change bugs
- It includes an access control list system and a number of auditing systems that are quite comprehensive to meet all sorts of demands
- It is able to interact with multiple operating systems and processor architectures

The grsecurity software is free and its development began in 2001, by first porting a number of security enhancing patches from the Openwall Project. It was first released for the 2.4.1 Linux kernel version and since then, development has continued. Over time, it included a PaX bundle of patches that offered the possibility of protecting memory pages. This is done by using a least-privilege approach, which implies that for the execution of a program, no more than the necessary actions should be taken with the help of extra or fewer steps.



If you're interested in finding more about PaX, you can access <http://en.wikipedia.org/wiki/PaX> and <https://pax.grsecurity.net/>.

Grsecurity has a number of features that are suitable mostly for web servers or servers that accept shell access from untrusted users. One of the major feature is the **Role-based Access Control (RBAC)**, which is the alternative to the already available **UNIX Discretionary Access Control (DAC)**, or even the latter, mandatory access control (MAC) that is offered by Smack or SELinux. The aim of RBAC is to offer a least privilege system in which the processes and users only have the minimum required privileges needed for archiving their tasks. One other feature that grsecurity has is related to the hardening of the `chroot()` system call to make sure that privilege escalation is eliminated. In addition to this, there are a number of miscellaneous features, such as auditing and `/proc` restrictions.

I took the liberty of keeping the features of the grsecurity defined in groups, as presented on the grsecurity website. They have been presented in the chapter because I think that knowing its features will help users and developers make the right decision when a security solution is required for their activities. A list with all the grsecurity features is mentioned as follows:

- Memory corruption defences:
 - Automatic response to brute force exploits
 - Hardened BPF JIT against spray attacks
 - Hardened userland memory permission
 - Random padding between thread stacks
 - Preventing direct userland access by a kernel
 - Industry leading ASLR
 - Bound checking kernel copies to/from a userland
- Filesystem Hardening:
 - Chroot hardening
 - Eliminating side-channel attacks against admin terminals
 - Preventing users from tricking Apache into accessing other user files
 - Hiding the processes of other users from unprivileged users
 - Providing trusted path execution
- Miscellaneous protections:
 - Preventing process snooping based on ptrace
 - Preventing the dumping of unreadable binaries

- Preventing attackers from autoloading vulnerable kernel modules
- Denying access to overly permissive IPC objects
- Enforcing consistent multithreaded privileges
- RBAC:
 - Intuitive design
 - Automatic full system policy learning
 - Automated policy analysis
 - Human-readable policies and logs
 - Stackable with LSM
 - Unconventional features
- GCC plugins:
 - Preventing integer overflows in size arguments
 - Preventing the leakage of stack data from previous syscalls
 - Adding entropy during early boot and runtime
 - Randomizing kernel structure layout
 - Making read-only sensitive kernel structures
 - Ensuring all kernel function pointers point to the kernel

Keeping the features of grsecurity in mind, we can now move towards the installation phase of grsecurity and its administrator called `gradm`.

The first thing that needs to be done is to get the corresponding packages and patches. As shown in the following command lines, the kernel version for which grsecurity is enabled is 3.14.19:

```
 wget https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.14.19.tar.gz
 wget https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.14.19.tar.sign
 wget http://grsecurity.net/stable/gradm-3.1-201502222102.tar.gz
 wget http://grsecurity.net/stable/gradm-3.1-201502222102.tar.gz.sig
 wget http://grsecurity.net/stable/grsecurity-3.1-3.14.36-201503182218.
 patch
 wget http://grsecurity.net/stable/grsecurity-3.1-3.14.36-201503182218.
 patch.sig
```

Security

After the packages are available, their signature needs to be checked. The signature check process for the Linux kernel is big and different from other systems, as follows:

```
wget http://grsecurity.net/spender-gpg-key.asc
sudo gpg --import spender-gpg-key.asc
sudo gpg --verify gradm-3.1-20150222102.tar.gz.sig
sudo gpg --verify grsecurity-3.1-3.14.35-201503092203.patch.sig
gzip -d linux-3.14.19.tar.gz
sudo gpg --verify linux-3.14.19.tar.sign
```

The first time this command is called, the signature is not verified, but the ID field is made available for later use. It is used to identify the public key from the PGP keyserver:

```
gpg: Signature made Mi 17 sep 2014 20:20:53 +0300 EEST using RSA key ID
6092693E
sudo gpg --keyserver hkp://keys.gnupg.net --recv-keys 6092693E
sudo gpg --verify linux-3.14.19.tar.sign
```

After all the packages are available and properly verified, we can now move to the kernel configuration phase. The first step is the patching process, which is done with the grsecurity patch, but this requires access to the Linux kernel source code first:

```
tar xf linux-3.14.19.tar
cd linux-3.14.19/
patch -p1 < ../grsecurity-3.1-3.14.35-201503092203.patch
```

In the patching process, include/linux/compiler-gcc5.h is missing from the source code, so this part of the patch requires skipping. However, after this, the patching process is finished without problems. With this step completed, the configuration phase can continue. There are generic configurations that should work without any extra modifications, but for each distribution there would always be some specific configuration available. To go through them and make sure that each one of them matches with your hardware, the following command can be used:

```
make menuconfig
```

If you are calling it for the first time, the preceding command has a warning message that will prompt you with the following:

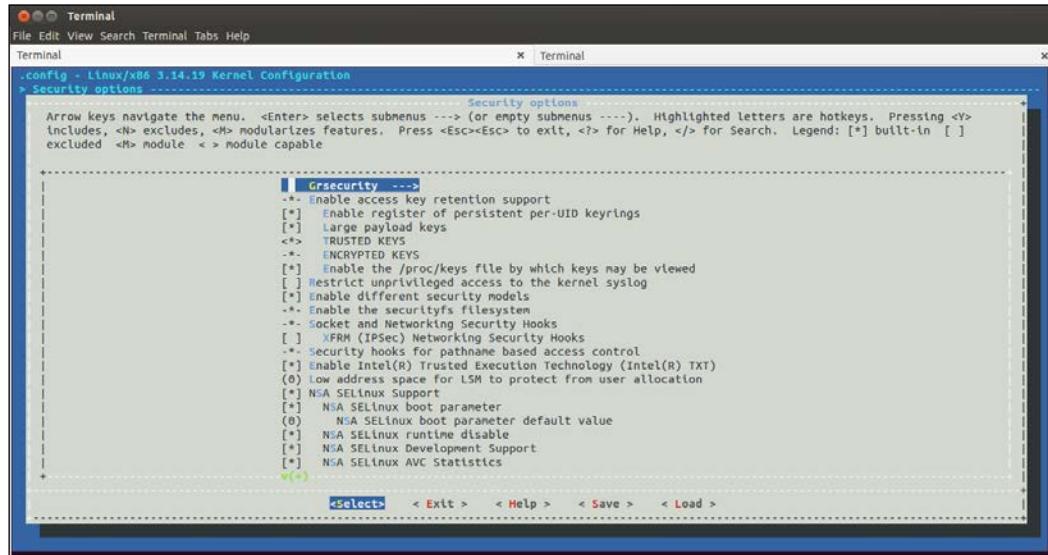
```
HOSTCC  scripts/basic/fixdep
HOSTCC  scripts/kconfig/conf.o
*** Unable to find the ncurses libraries or the
*** required header files.
```

```
*** 'make menuconfig' requires the ncurses libraries.
***
*** Install ncurses (ncurses-devel) and try again.
***
make[1]: *** [scripts/kconfig/dochecklxdialog] Error 1
make: *** [menuconfig] Error 2
```

It can be solved by installing the libncurses5-dev package, using the following command:

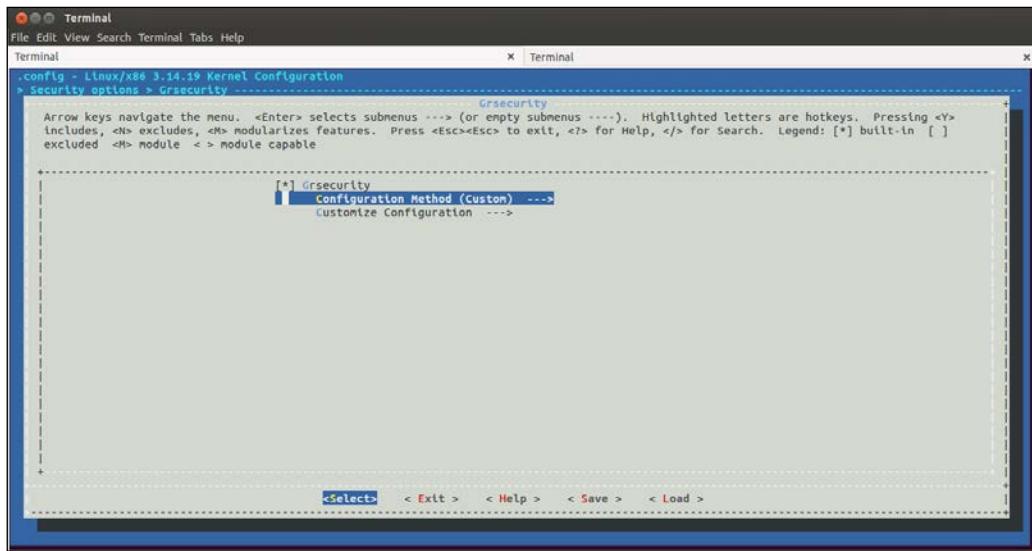
```
sudo apt-get install libncurses5-dev
```

With these problems fixed, the configuration process can continue. The grsecurity option is available inside the security option subsection, as depicted in the following screenshot:

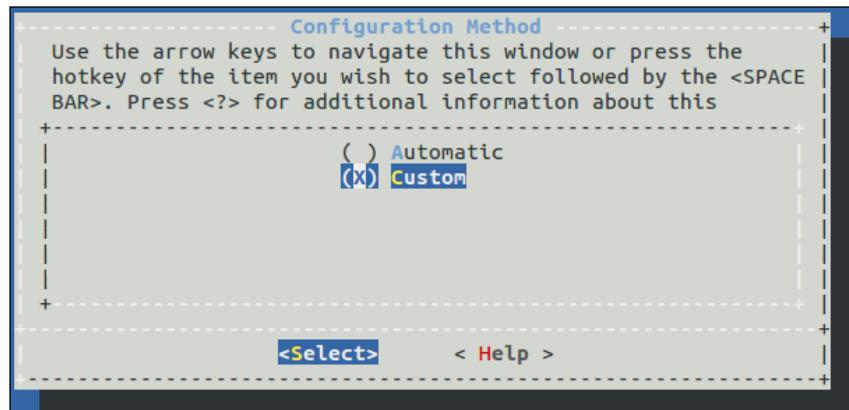


Security

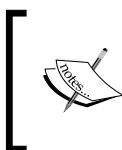
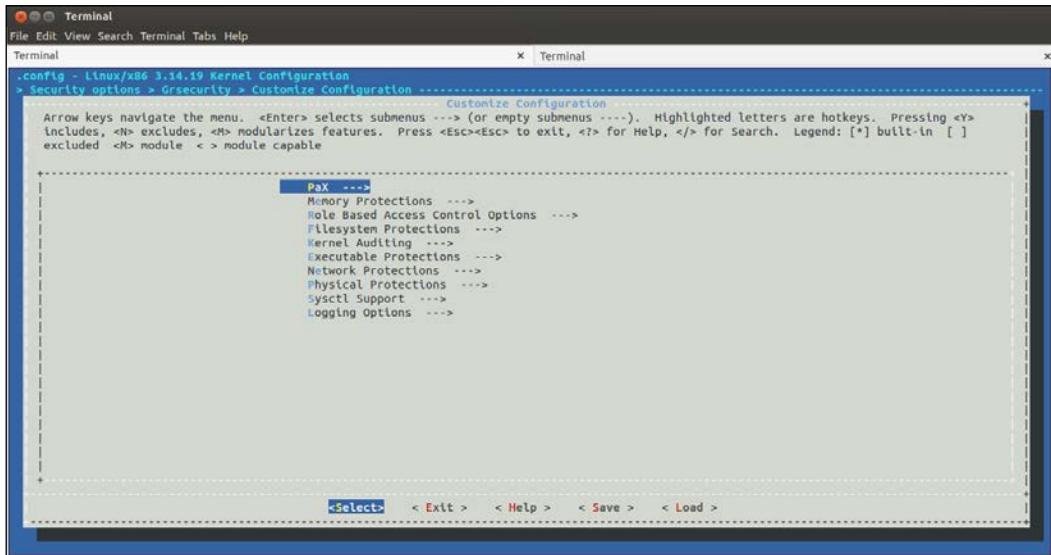
Inside the grsecurity option, there are two more submenu options. More details about this can be seen in the following screenshot:



The first option refers to the configuration method, which can be **Custom** or **Automatic**:



The second option refers to the actual available configuration options:



More information about Grsecurity and the PaX configuration options can be found at http://en.wikibooks.org/wiki/Grsecurity/Appendix/Grsecurity_and_PaX_Configuration_Options.

One piece of advice I would like to offer is that first enable the **Automatic** configuration method and then proceed with the Custom configuration to fine tune the Grsecurity and PaX settings if necessary. Another tip would be to enable the **Grsecurity | Customize Configuration | Sysctl Support** option because it offers the possibility of changing the grsecurity options without compiling the kernel again. Of course, if the **Automatic** configuration method is selected, this option is enabled by default. The auditing option produces a big number of logs, so to prevent log flooding, make sure that **Grsecurity | Customize Configuration | Logging Options** is also enabled.

The next tool from the grsecurity family is the `gradm` administrator, which is a powerful parser for ACLs and also optimizes them. To make sure that this utility can be installed, the installation process requires that the host operating machine for `gradm` offers grsecurity support or else the compilation process will fail. There are also a number of other packages that are required before installing `gradm`: `lex`, `flex`, `byacc`, `bison`, and even `pam`, if necessary.

Once all the dependencies are met, the installation process can start. One last bit of information I'd like to give you is that if the distribution that you use comes with a kernel that has support for grsecurity patches, then you may first want to check it because the patches can also come with the `gradm` utility pre-installed.

[More information about the Grsecurity administration can be found at the following links:

 http://en.wikibooks.org/wiki/Grsecurity/The_Administration.Utility

http://en.wikibooks.org/wiki/Grsecurity/Additional_Utils

http://en.wikibooks.org/wiki/Grsecurity/Runtime_Configuration

]

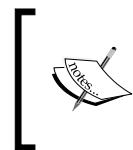
Inside the Yocto layers, there is support for the `gradm` recipe that is inside the `meta-oe` layer. It is available at `recipes-support/gradm/gradm_3.0.bb` on the master branch. Also, a grsecurity kernel configuration is available on the master branch for the `meta-accel` layer; the exact location for the configuration fragment is `recipes-kernel/linux/linux-yocto-ii0/grsec.cfg`. For anyone interested in learning about the concrete grsecurity support provided in Yocto, I believe the road is clear for you to start working on such a thing. One piece of advice, though, you should first ask the Yocto Project community whether anyone has started doing this already.

Security for the Yocto Project

In the Yocto Project, the security question is still young. Since this project was announced less than five years ago, it is only normal that discussions about security started in the last year or so. There is, of course, a specialized mailing list for the security team and it includes a large number of individuals from various companies, but their working procedure is not quite finished since it's currently in state of work in progress.

The activities that are mainly realized by the members of the security team consist of being aware of the latest and most dangerous security threats and making sure that they find the fixes, even if it includes fixing themselves and applying the changes inside Yocto's available layers.

For the time being, the most time consuming of the security activity revolves around the Poky reference system, but there are also initiatives taken by various companies to try to push a series of patches toward various BSP maintainer layers or other third-party layers. For those of you interested, the mailing list of security-related discussions is yocto-security@yoctoproject.org. Also, until the group is formed, they can be found in the #yocto IRC available at <http://webchat.freenode.net/?channels=#yocto>, or even at the Yocto technical team meeting that takes place once every two weeks.



More information about the security team can be found on their Wiki page. I encourage everyone interested in this subject to visit it at least once at <https://wiki.yoctoproject.org/wiki/Security>.



Meta-security and meta-selinux

In this section, the layer initiatives related to the security tools of Linux are presented. In this chapter, two layers that provide both security and hardening tools are available for the Linux kernel and its libraries. Their purpose is to simplify mode embedded devices, make sure that they're secure, and maybe offer the security level similar to a desktop.

Since embedded devices have become increasingly competent and powerful, concerns related to security can only be natural. The Yocto Project's initiative layers, here, I am referring to meta-security and meta-selinux, take another step in simplifying the process to ensure secure, hardened, and protected Linux systems. Together, with the detect and fix vulnerability system, they are implemented inside the security team, and help with the ideal of having the same level of security on embedded devices as desktops, along with taking this idea a step further. Having said this, let's proceed to the actual explanation of layers.

Meta-security

Inside the meta-security layer, there are tools that are used to secure, harden, and protect embedded devices that may offer exterior access to various entities. If the device is connected to the Internet or is susceptible to any form of attack or hijacking, then the meta-security layer may be the first stop for you. With this layer and the meta-selinux layer, the Yocto Project tries to provide security levels that are suitable for most of the community or embedded user devices. Of course, enhancing the support for various tools or adding new ones is not forbidden, so do not hesitate and add your contribution for enhancing tools if you feel the need or urge to do so. Any new commit or committer is welcome - our community is really friendly.

As you're already used to, the tools provided are open source packages that are suitable for embedded devices. Inside the meta-security layer a number of them are available, each one trying to offer not only system hardening, but also security checking, security, port scanning, and other useful features that target various levels of security. The following packages are included:

- Bastille
- Redhat-security
- Pax-utils
- Buck-security
- Libseccomp
- Ckecksecurity
- Nikto
- Nmap
- Clamav
- Isic
- Samhain
- Suricata
- Tripwire

Besides these packages, there are a number of libraries and also **TOMOYO**, a kernel security module for a MAC implementation, which is also very useful as a system analysis tool. It was first released in March 2003, and was sponsored by NTT Data Corporation, Japan, until March 2012.

TOMOYO's main focus is the system behavior. For this, every process involved in the creation of the system declares its behavior and the required resources necessary to achieve a purpose. It consists of two components: one kernel component, linux-ccs, and a user space one, ccs-tools; both are required for proper functionality. TOMOYO tries to provide a MAC implementation that is both practical and easy to use. Finally, it likes to let a system be usable for a majority of users, being perfect for average users and system administrators. It is different from SELinux because it has an automatic policy configuration mechanism offered by the **LEARNING mode**; also, its policy language is very easy to grasp.

After protection is enabled, TOMOYO Linux acts as a watchdog that restricts the processes from using more than what they had declared initially. Its main features include the following:

- System analysis
- Tools that offer aid in the process of policy generation
- Simple to use and understand syntax
- Easy to use
- Increased security of the system through the MAC implementation
- Contains a small number of dependencies (the embedded GNU C library, libncurses, and GNU readline library)
- No modification of the already available binaries inside the root filesystem
- Since the version 2.6.30, the Linux kernel merged with the TOMOYO kernel module, making only the enabling of the module in the configuration phase necessary. It started as a patch that provided MAC support, and the porting inside a mainline kernel required a redesign using hooks into the **LSM (Linux Security Modules)**, which also includes SELinux, AppArmor, and SMACK. However, since more hooks would be necessary for the integration of the remaining MAC functionalities, there are two other parallel development lines for the project, as follows:
- **TOMOYO Linux 1.x:** This is the original code version:
 - It uses nonstandard specific hooks
 - It offers all the MAC features
 - It is released as a patch for the kernel since it does not depend on LSM
 - Its latest version is 1.7.1

- **TOMOYO Linux 2.x:** This is the mainline source code version:
 - It uses standard LSM hooks
 - It contains a fewer subset of features
 - It is an integral component of the 2.6.30 Linux kernel version
 - The latest version is 2.5.0 and offers support for Linux kernel version 3.2
- **AKARI and TOMOYO 1.x fork version:**
 - It also uses standard LSM hooks
 - It is characterized by having a fewer set of features compared to TOMOYO 1.x but not with TOMOYO 2.x
 - It is released as LSM; no recompilation of the kernel is necessary



For those of you interested in a comparison between the three versions, refer to <http://akari.sourceforge.jp/comparison.html.en>.



The next package is `samhain`, a system integrity monitoring and reporting tool used by system administrators that suspect changes or activities on their systems. Its operation is based on a client/server environment and is able to monitor multiple hosts while providing a centralized maintenance and logging system. Besides the already advertised functionalities, it is also able to provide port monitoring, detection of rogue SUID, rootkit detection, and also hidden processes that add to the fact that it offers support for multiple platforms; it is a really interesting tool to have.

The next element here falls in the same category as `samhain` and it is called `tripwire`. It is another integrity tool, but this one tries to detect changes for filesystem objects and works as a host intrusion detection system. Information is stored in a database after each file scan and the results are compared with the already available results. Any changes that are made are reported back to the user.

Bastille is a hardening program used to secure the environment and system for a Unix host. It uses rules to accomplish its goals and does this by first calling the `bastille -c` command that makes you pass through a long list of questions. After they are answered, a configuration file is created and executed and this symbolizes the fact that your operating system is now hardened according to your needs. If a configuration file is already available on the system by calling `bastille -b`, it can be set up for system hardening.

The next tool is `redhat-security`, which is a collection of scripts used for various problems related to security scanning. The following are a collection of the tools needed to run a `redhat-security` script to simply invoke one script in the terminal:

- `find-chroot.sh`: This tool scans the whole system for ELF files that call `chroot` and also include a call to `chdir`. The programs that fail this test do not contain `cwd` inside `chroot` and they are not protected and safe to use.
- `find-chroot-py.sh`: This tool is similar to the preceding point, but only tests Python scripts.
- `rpm-chksec.sh`: This tool takes an `rpm` file and checks its content for its compiling flags. It does this for security reasons. If the results are green, then everything is OK, yellow means passable, and red requires the user's attention.
- `find-nodrop-groups.sh`: This tool scans the whole system for programs that change the UID or GID without calling the `setgroups` and `initgroups` calls.
- `rpm-drop-groups.sh`: This tool scans the whole system similar to the preceding tool, but this one uses the available RPM files.
- `find-execstack.sh`: This tool scans the whole system for ELF files that mark the stack as executable. It is used to identify programs that are susceptible to stack buffer overflow.
- `find-sh4errors.sh`: This tool scans the whole system for shell scripts and checks their correctness by using the `sh -n` command.
- `find-hidden-exec.sh`: This tool scans the system for hidden executables and reports the results back to the user for investigation.
- `selinux-ls-unconfined.sh`: This tool is used to scan all the running processes and look for the `initrc_t` label or `inetd` on them (this means that they are daemons that are running unconfined). The problems should be reported as SELinux policy problems.
- `selinux-check-devides.sh`: This tool checks all the available devices too see if they are correctly labelled. It is also marked as a SELinux policy problem that should be solved.
- `find-elf4tmp.sh`: This tool scans the whole system and checks whether the used `tmp` files are well known, are created with `mkttemp`, or have some obscure format.

- `find-sh3tm.sh`: This tool also scans the filesystem, although only inside `/tmp` and looks for ELF files there. When it finds them, it checks if any of the random name generators function was called on them by investigating the symbol table. If the result is affirmative, it will output the string value.
- `lib-bin-check.sh`: This tool checks the packages of libraries and their the package they contain. It is based on the idea that the fewer binaries available on a system, the more secure it is.

Another tool that is included is `pax-utils`. It also includes a number of scripts that scan ELF binaries mostly for consistency, but this is not all. Take a look at some of them:

- `scanelf`: This tool is used to find pre-information about the ELF structure of the binary
- `dumpelf`: This tool is a user space utility used to dump the internal ELF structure in equivalent C structures for debugging or reference purposes
- `pspax`: This tool is used to scan `/proc` and list the various ELF types available and their corresponding PaX flags, attributes, and filenames

Now, the next tool that will be presented is a security scanner that is different from the already presented `bastille`. Similar to the `redhat-security` command, this one also executes a number of scripts and can be configured to confirm the user's needs. It is suitable for Debian and Ubuntu users, and before calling the `buck-security` executable, there are a few configurations that need to be done. Use `export GPG_TTY='`tty`'` to make sure that all the functionalities of the `buck-security` are enabled and before executing the tool, check inside the `conf/buck-security.conf` configuration file to check that your needs are fulfilled.

Suricata is a high-performance IDS/IPS and Security Monitoring engine for the network. It is owned and maintained by **OISF (Open Information Security Foundation)** and its supporters. It uses the **HTP** library that is a very powerful HTTP parser and normalizer and offers some nice features, such as protocol identification, MD5 checksum, file identification, and even extraction.

ISIC, on the other hand, is what its name suggests, an IP Stack Integrity Checker. It is, in fact, a suite of utilities for IP Stack ad other stacks, such as TCP, ICMP, UDP, and others that test either the firewall, or the protocol itself.

For any web server, **nikto** is the tool to execute on your device. It is a scanner used to run a suite of tests that identifies dangerous CGI1s or other files. It also presents an outdated version for more than 1250 servers and various lists of vulnerabilities for each version.

Next on the list is the **libseccomp** library that provides an easy-to-use abstract interface to the Linux kernel, `syscall`, filtering a mechanism called `seccomp`. It does this by abstracting the BPF `syscall` filter language and presenting it a more user-friendly format for application developers in general.

Checksecurity is the next package on the line which uses a collection of shell scripts and other plugins for testing various changes to `setuid` programs. Using the filter defined in `/etc/checksecurity.conf`, it scans the mounted filesystems and compares the already available list of `setuid` programs to the newly scanned ones and prints the changes for the user to see. It also offers information about these filesystems that were mounted unsecure.

ClamAV is an antivirus for Unix that operates from the command line. It is a very good engine for tracking trojans, malware, viruses, and detection of other malicious threats. It can do a large variety of things from e-mail scanning to web scanning and end-point security. It also has a very versatile and scalable daemon, command-line scanner, and database interaction tool.

The last on the list is **Network Mapper (nmap)**. It is the most well known and is used for security auditing as well as a network discovery tool by network and system administrators. It is used to manage service upgrade schedules, network inventory, monitoring various services, or even host uptime.

These are the tools supported and offered inside the meta-security layer. I took the liberty of presenting most of them in a succinct manner with the purpose of making them available to you in an easy fashion. It is my opinion that for security problems, one should not overcomplicate things and only keep the solutions that fit your needs best. By presenting a large palette of tools and software components, I tried to do two things: make a larger number of tools available for the general public and also help you make a decision with regard to the tools that might help you in your quest to offer and even maintain a secure system. Of course, curiosity is encouraged, so make sure that you check out any other tools that might help you on your quest to find out more about security, and why they should not be integrated inside the meta-security layer.

Meta-selinux

The other available security layer is represented by the meta-selinux layer. This one is different from meta-security because it only offers support for one tool, but as mentioned in the preceding tool, it is so big and vast that it spreads its wings across the whole system.

The layer's purpose is to enable the support for SELinux and offer it through Poky to anyone in the Yocto Project community for use if required. As mentioned previously, since it influences the whole Linux system, most of the work on this layer is done inside the bbappend files. I hope you enjoy working with the functionalities available inside this layer and maybe even contribute to it if you see fit.

This layer not only contains a number of impressive bbappend files, but also offers a list of packages that could be used not only as SELinux extensions. These packages can be used also for other self-contained purposes too. The available packages inside the meta-selinux layer are as follows:

- audit
- libcap-ng
- setools
- swig
- ustr

I will start the introduction of this layer with the **audit** userspace tool, which as the name suggests, is a tool that can be used for auditing, more specifically for kernel auditing. It uses a number of utilities and libraries to search and store recorded data. The data is generated through an audit subsystem available inside the Linux kernel. It is designed to work as a standalone component, but it cannot offer **Common Criteria (CC)** or **FIPS 140-2** functionalities without a second security component being available.

The next element on the list is **libcap-ng**, an alternative library with simplified POSIX capabilities that can be compared to the traditional libcap solution. It offers utilities that analyze running applications and print out their capabilities, or if they have an open ended bounding set. For an open bounding set that lacks the `securebit`, `NOROOT` flag will permit only by using an `execve()` call to retain full capabilities for applications that retain the `0` UID. By using the libcap-ng libraries, these applications that have the most privileges, are very easy to spot and deal with tools. The interaction and their detection is done with other tools, such as **netcap**, **pscap**, or **filecap**.

SETools is a policy analysis tool. It is in fact, an extension of SELinux and contains a collection of libraries, graphical tools, and command-lines that try simply analyze the SELinux policies. The primary tools that this open source project are as follows:

- `apol`: This is a tool used to analyze SELinux policies
- `sediff`: This acts as a semantic differentiator between SELinux policies
- `seaudit`: This is a tool used to analyze audit messages for SELinux
- `seaudit-report`: This is used to generate a highly customizable audit report based on available audit logs
- `sechecker`: This is a command-line tool that is engaged in modular checks of SELinux policies
- `secmds`: This is another command-line tool that is used to reach and analyze SELinux policies

Next is **SWIG (Simplified Wrapper and Interface Generator)**, a software development tool used with a variety of target languages to create a high-level programming environment, user interfacing, and anything else that is necessary. It is usually used for fast testing or prototyping because it generates the glue that a target language can call inside the C or C++ code.

The last component to be presented is a micro string API for a C language called **ustr**, which has the benefit of how overheads compared to available APIs. It is very easy to use in the C code as it only includes a header file and is ready for usage. Its overhead over `strdup()` for strings varies from 85.45 for 1-9 byte strings to 23.85 for 1-198 byte strings. For a simpler example, if an 8 byte storage `ustr` uses 2 bytes, the `strdup()` function uses 3 bytes.

This is where other tools and libraries are available alongside the SELinux functionality, although some of them can be used as separate components or in tandem with other available software components that were presented here. This would add more value to the SELinux product, so it only seems fair to find them in the same place.

For those of you interested in obtaining a SELinux enhance distribution, you could choose to use one of the two available images in the meta-selinux layer: `core-image-selinux-minimal.bb` or `core-image-selinux.bb`. The alternative would be to incorporate one of the available SELinux-specific defined package groups, `packagegroup-selinux-minimal` or `packagegroup-core-selinux`, into a newly defined image according to the needs of a developer. After this choice is made and the configuration is done accordingly, the only thing remaining would be to call `bitbake` for the chosen image and at the end of the build process, a custom Linux distribution will reveal itself with SELinux support enabled and can be tweaked some more if necessary.

Summary

In this chapter, you were presented with information about both kernel-specific security projects as well as external projects. Most of these were presented in a bad manner. You were also given information related to how various security subsystems and subgroups are keeping pace with various security threats and security project implementations.

In the next chapter, we will move on to another interesting subject. Here, I am referring to the virtualization area. You will find more about the meta-virtualization aspect later along with various virtualization implementations, such as KVM, which has gathered a huge track over the last few years and has established itself as a standard. I will let the other elements, which will be presented in the next chapter, be a secret. Let's now further explore the content of this book.

12

Virtualization

In this chapter, you will be presented with information about various concepts that appeared in the Linux virtualization section. As some of you might know, this subject is quite vast and selecting only a few components to be explained is also a challenge. I hope my decision would please most of you interested in this area. The information available in this chapter might not fit everyone's need. For this purpose, I have attached multiple links for more detailed descriptions and documentation. As always, I encourage you to start reading and finding out more, if necessary. I am aware that I cannot put all the necessary information in only a few words.

In any Linux environment today, Linux virtualization is not a new thing. It has been available for more than ten years and has advanced in a really quick and interesting manner. The question now does not revolve around virtualization as a solution for me, but more about what virtualization solutions to deploy and what to virtualize.

There are, of course, scenarios in which virtualization is not a solution. In embedded Linux, there are a large category of domains for which virtualization does not apply, mostly because some workloads are a better fit on top of hardware. However, for others that do not have these kind of requirements, there are quite a few advantages to using virtualization. More information about the various virtualization strategies, cloud computing, and other related topics will be discussed in this chapter, so let's have a look.

Linux virtualization

The first benefit everyone sees when looking at virtualization is the increase in server utilization and the decrease in energy costs. Using virtualization, the workloads available on a server are maximized, which is very different from scenarios where hardware uses only a fraction of the computing power. It can reduce the complexity of interaction with various environments and it also offers an easier-to-use management system. Today, working with a large number of virtual machines is not as complicated as interaction with a few of them because of the scalability most tools offer. Also, the time of deployment has really decreased. In a matter of minutes, you can deconfigure and deploy an operating system template or create a virtual environment for a virtual appliance deploy.

One other benefit virtualization brings is flexibility. When a workload is just too big for allocated resources, it can be easily duplicated or moved on another environment that suit its needs better on the same hardware or on a more potent server. For a cloud-based solution regarding this problem, the sky is the limit here. The limit may be imposed by the cloud type on the basis of whether there are tools available for a host operating system.

Over time, Linux was able to provide a number of great choices for every need and organization. Whether your task involves server consolidation in an enterprise data centre, or improving a small nonprofit infrastructure, Linux should have a virtualization platform for your needs. You simply need to figure out where and which project you should chose.

Virtualization is extensive, mainly because it contains a broad range of technologies, and also since large portions of the terms are not well defined. In this chapter, you will be presented with only components related to the Yocto Project and also to a new initiative that I personally am interested in. This initiative tries to make **Network Function Virtualization (NFV)** and **Software-Defined Networking (SDN)** a reality and is called **Open Platform for NFV (OPNFV)**. It will be explained here briefly.

SDN and NFV

I have decided to start with this topic because I believe it is really important that all the research done in this area is starting to get traction with a number of open source initiatives from all sorts of areas and industries. Those two concepts are not new. They have been around for 20 years since they were first described, but the last few years have made possible it for them to resurface as real and very possible implementations. The focus of this section will be on the *NFV* section since it has received the most amount of attention, and also contains various implementation proposals.

NFV

NFV is a network architecture concept used to virtualize entire categories of network node functions into blocks that can be interconnected to create communication services. It is different from known virtualization techniques. It uses **Virtual Network Functions (VNF)** that can be contained in one or more virtual machines, which execute different processes and software components available on servers, switches, or even a cloud infrastructure. A couple of examples include virtualized load balancers, intrusion detected devices, firewalls, and so on.

The development product cycles in the telecommunication industry were very rigorous and long due to the fact that the various standards and protocols took a long time until adherence and quality meetings. This made it possible for fast moving organizations to become competitors and made them change their approach.

In 2013, an industry specification group published a white paper on software-defined networks and OpenFlow. The group was part of **European Telecommunications Standards Institute (ETSI)** and was called Network Functions Virtualisation. After this white paper was published, more in-depth research papers were published, explaining things ranging from terminology definitions to various use cases with references to vendors that could consider using NFV implementations.

ETSI NFV

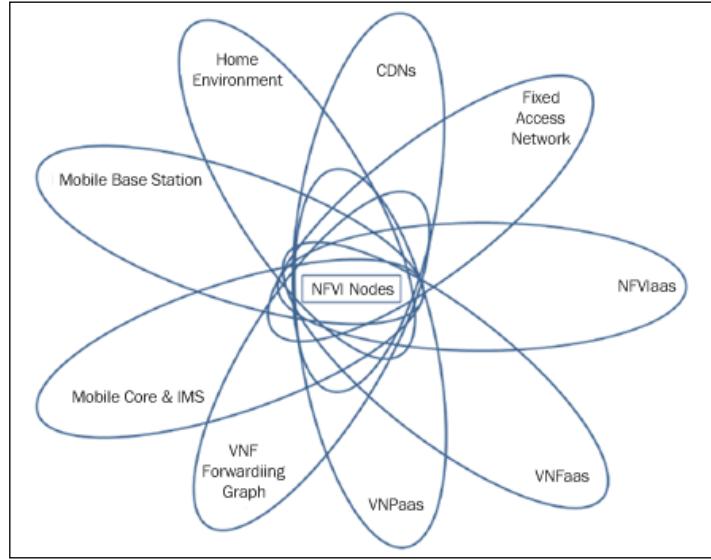
The ETSI NFV workgroup has appeared useful for the telecommunication industry to create more agile cycles of development and also make it able to respond in time to any demands from dynamic and fast changing environments. SDN and NFV are two complementary concepts that are key enabling technologies in this regard and also contain the main ingredients of the technology that are developed by both telecom and IT industries.

The NFV framework consist of six components:

- **NFV Infrastructure (NFVI):** It is required to offer support to a variety of use cases and applications. It comprises of the totality of software and hardware components that create the environment for which VNF is deployed. It is a multitenant infrastructure that is responsible for the leveraging of multiple standard virtualization technologies use cases at the same time. It is described in the following **NFV Industry Specification Groups (NFV ISG)** documents:
 - NFV Infrastructure Overview
 - NFV Compute

- NFV Hypervisor Domain
- NFV Infrastructure Network Domain

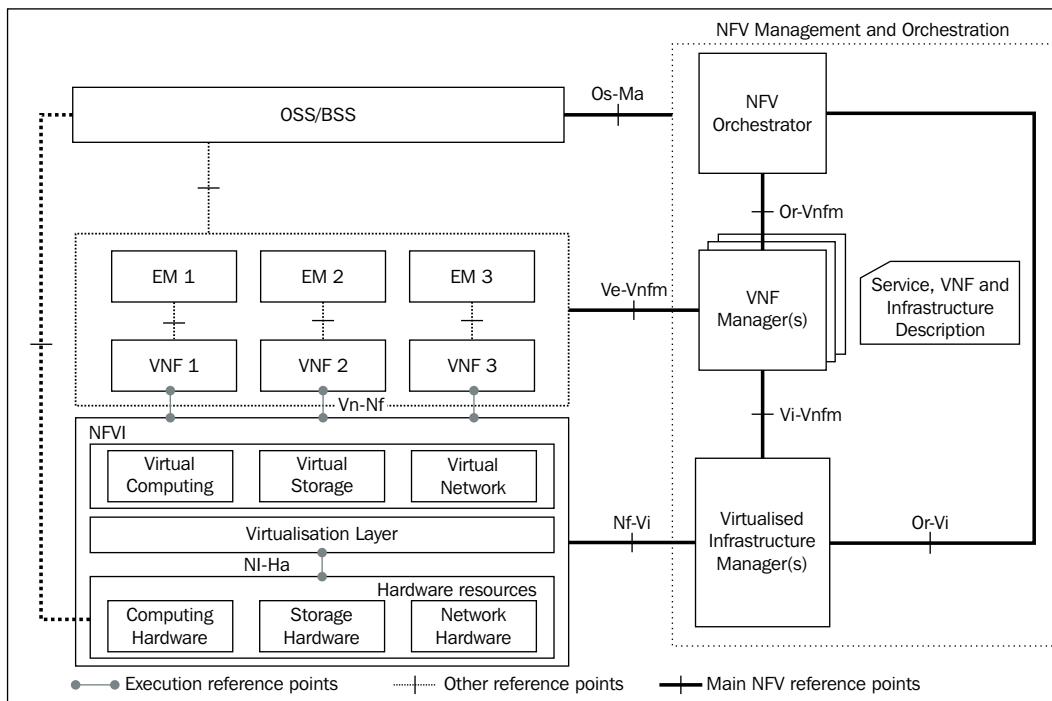
The following image presents a visual graph of various use cases and fields of application for the NFV Infrastructure.



- **NFV Management and Orchestration (MANO):** It is the component responsible for the decoupling of the compute, networking, and storing components from the software implementation with the help of a virtualization layer. It requires the management of new elements and the orchestration of new dependencies between them, which require certain standards of interoperability and a certain mapping.
- **NFV Software Architecture:** It is related to the virtualization of the already implemented network functions, such as proprietary hardware appliances. It implies the understanding and transition from a hardware implementation into a software one. The transition is based on various defined patterns that can be used in a process.
- **NFV Reliability and Availability:** These are real challenges and the work involved in these components started from the definition of various problems, use cases, requirements, and principles, and it has proposed itself to offer the same level of availability as legacy systems. It relates to the reliability component and the documentation only sets the stage for future work. It only identifies various problems and indicates the best practices used in designing resilient NFV systems.

- **NFV Performance and Portability:** The purpose of NFV, in general, is to transform the way it works with networks of future. For this purpose, it needs to prove itself as wordy solution for industry standards. This section explains how to apply the best practices related to performance and portability in a general VNF deployment.
- **NFV Security:** Since it is a large component of the industry, it is concerned about and also dependent on the security of networking and cloud computing, which makes it critical for NFV to assure security. The Security Expert Group focuses on those concerns.

An architectural of these components is presented here:



After all the documentation is in place, a number of proof of concepts need to be executed in order to test the limitation of these components and accordingly adjust the theoretical components. They have also appeared to encourage the development of the NFV ecosystem.



For more information about the available roof of concepts and specifications for NFV, refer to these links: <http://www.etsi.org/technologies-clusters/technologies/nfv/nfv-poc?tab=2> and <http://www.etsi.org/technologies-clusters/technologies/nfv>.

SDN

Software-Defined Networking (SDN) is an approach to networking that offers the possibility to manage various services using the abstraction of available functionalities to administrators. This is realized by decoupling the system into a control plane and data plane and making decisions based on the network traffic that is sent; this represents the control plane realm, and where the traffic is forwarded is represented by the data plane. Of course, some method of communication between the control and data plane is required, so the OpenFlow mechanism entered into the equation at first; however other components could as well take its place.

The intention of SDN was to offer an architecture that was manageable, cost-effective, adaptable, and dynamic, as well as suitable for the dynamic and high-bandwidth scenarios that are available today. The OpenFlow component was the foundation of the SDN solution. The SDN architecture permitted the following:

- **Direct programming:** The control plane is directly programmable because it is completely decoupled by the data plane.
- **Programmatically configuration:** SDN permitted management, configuration, and optimization of resources through programs. These programs could also be written by anyone because they were not dependent on any proprietary components.
- **Agility:** The abstraction between two components permitted the adjustment of network flows according to the needs of a developer.
- **Central management:** Logical components could be centered on the control plane, which offered a viewpoint of a network to other applications, engines, and so on.

- **Opens standards and vendor neutrality:** It is implemented using open standards that have simplified the SDN design and operations because of the number of instructions provided to controllers. This is smaller compared to other scenarios in which multiple vendor-specific protocols and devices should be handled.

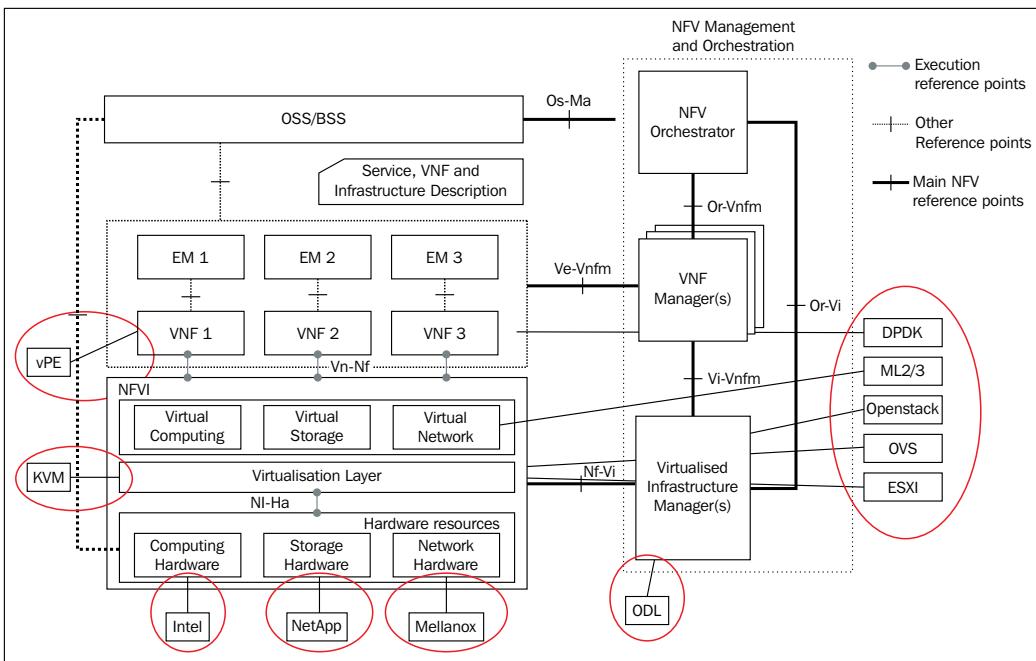
Also, meeting market requirements with traditional solutions would have been impossible, taking into account newly emerging markets of mobile device communication, Internet of Things (IoT), Machine to Machine (M2M), Industry 4.0, and others, all require networking support. Taking into consideration the available budgets for further development in various IT departments, were all faced to make a decision. It seems that the mobile device communication market all decided to move toward open source in the hope that this investment would prove its real capabilities, and would also lead to a brighter future.

OPNFV

The Open Platform for the NFV Project tries to offer an open source reference platform that is carrier-graded and tightly integrated in order to facilitate industry peers to help improve and move the NFV concept forward. Its purpose is to offer consistency, interoperability, and performance among numerous blocks and projects that already exist. This platform will also try to work closely with a variety of open source projects and continuously help with integration, and at the same time, fill development gaps left by any of them.

This project is expected to lead to an increase in performance, reliability, serviceability, availability, and power efficiency, but at the same time, also deliver an extensive platform for instrumentation. It will start with the development of an NFV infrastructure and a virtualized infrastructure management system where it will combine a number of already available projects. Its reference system architecture is represented by the x86 architecture.

The project's initial focus point and proposed implementation can be consulted in the following image. From this image, it can be easily seen that the project, although very young since it was started in November 2014, has had an accelerated start and already has a few implementation propositions. There are already a number of large companies and organizations that have started working on their specific demos. OPNFV has not waited for them to finish and is already discussing a number of proposed project and initiatives. These are intended both to meet the needs of their members as well as assure them of the reliability various components, such as continuous integration, fault management, test-bed infrastructure, and others. The following figure describes the structure of OPNFV:



The project has been leveraging as many open source projects as possible. All the adaptations made to these project can be done in two places. Firstly, they can be made inside the project, if it does not require substantial functionality changes that could cause divergence from its purpose and roadmap. The second option complements the first and is necessary for changes that do not fall in the first category; they should be included somewhere in the OPNFV project's codebase. None of the changes that have been made should be up streamed without proper testing within the development cycle of OPNFV.

Another important element that needs to be mentioned is that OPNFV does not use any specific or additional hardware. It only uses available hardware resources as long the VI-Ha reference point is supported. In the preceding image, it can be seen that this is already done by having providers, such as Intel for the computing hardware, NetApp for storage hardware, and Mellanox for network hardware components.

The OPNFV board and technical steering committee have a quite large palette of open source projects. They vary from **Infrastructure as a Service (IaaS)** and hypervisor to the SDN controller and the list continues. This only offers the possibility for a large number of contributors to try some of the skills that maybe did not have the time to work on, or wanted to learn but did not have the opportunity to. Also, a more diversified community offers a broader view of the same subject.

There are a large variety of appliances for the OPNFV project. The virtual network functions are diverse for mobile deployments where mobile gateways (such as Serving Gateway (SGW), Packet Data Network Gateway (PGW), and so on) and related functions (Mobility Management Entity (MME) and gateways), firewalls or application-level gateways and filters (web and e-mail traffic filters) are used to test diagnostic equipment (Service-Level Agreement (SLA) monitoring). These VNF deployments need to be easy to operate, scale, and evolve independently from the type of VNF that is deployed. OPNFV sets out to create a platform that has to support a set of qualities and use-cases as follows:

- A common mechanism is needed for the life-cycle management of VNFs, which include deployment, instantiation, configuration, start and stop, upgrade/downgrade, and final decommissioning
- A consistent mechanism is used to specify and interconnect VNFs, VNFCs, and PNFs; these are independant of the physical network infrastructure, network overlays, and so on, that is, a virtual link
- A common mechanism is used to dynamically instantiate new VNF instances or decommission sufficient ones to meet the current performance, scale, and network bandwidth needs
- A mechanism is used to detect faults and failure in the NFVI, VIM, and other components of an infrastructure as well as recover from these failures
- A mechanism is used to source/sink traffic from/to a physical network function to/from a virtual network function
- NFVI as a Service is used to host different VNF instances from different vendors on the same infrastructure

There are some notable and easy-to-grasp use case examples that should be mentioned here. They are organized into four categories. Let's start with the first category: the Residential/Access category. It can be used to virtualize the home environment but it also provides fixed access to NFV. The next one is data center: it has the virtualization of CDN and provides use cases that deal with it. The mobile category consists of the virtualization of mobile core networks and IMS as well as the virtualization of mobile base stations. Lastly, there are cloud categories that include NFVIaaS, VNFaaS, the VNF forwarding graph (Service Chains), and the use cases of VNPAaaS.



More information about this project and various implementation components is available at <https://www.opnfv.org/>. For the definitions of missing terminologies, please consult http://www.etsi.org/deliver/etsi_gs/NFV/001_099/003/01.02.01_60/gs_NFV003v010201p.pdf.

Virtualization support for the Yocto Project

The meta-virtualization layer tries to create a long and medium term production-ready layer specifically for an embedded virtualization. This roles that this has are:

- Simplifying the way collaborative benchmarking and researching is done with tools, such as KVM/LxC virtualization, combined with advance core isolation and other techniques
- Integrating and contributing with projects, such as OpenFlow, OpenvSwitch, LxC, dmtcp, CRIU and others, which can be used with other components, such as OpenStack or Carrier Graded Linux.

To summarize this in one sentence, this layer tries to provide support while constructing OpenEmbedded and Yocto Project-based virtualized solutions.

The packages that are available in this layer, which I will briefly talk about are as follows:

- CRIU
- Docker
- LXC
- Irbalance

- Libvirt
- Xen
- Open vSwitch

This layer can be used in conjunction with the `meta-cloud-services` layer that offer cloud agents and API support for various cloud-based solutions. In this section, I am referring to both these layers because I think it is fit to present these two components together. Inside the `meta-cloud-services` layer, there are also a couple of packages that will be discussed and briefly presented, as follows:

- openLDAP
- SPICE
- Qpid
- RabbitMQ
- Tempest
- Cyrus-SASL
- Puppet
- oVirt
- OpenStack

Having mentioned these components, I will now move on with the explanation of each of these tools. Let's start with the content of the meta-virtualization layer, more exactly with `CRU` package, a project that implements **Checkpoint/Restore In Userspace** for Linux. It can be used to freeze an already running application and checkpoint it to a hard drive as a collection of files. These checkpoints can be used to restore and execute the application from that point. It can be used as part of a number of use cases, as follows:

- **Live migration of containers:** It is the primary use case for a project. The container is check pointed and the resulting image is moved into another box and restored there, making the whole experience almost unnoticeable by the user.
- **Upgrading seamless kernels:** The kernel replacement activity can be done without stopping activities. It can be check pointed, replaced by calling `kexec`, and all the services can be restored afterwards.
- **Speeding up slow boot services:** It is a service that has a slow boot procedure, can be check pointed after the first start up is finished, and for consecutive starts, can be restored from that point.

- **Load balancing of networks:** It is a part of the `TCP_REPAIR` socket option and switches the socket in a special state. The socket is actually put into the state expected from it at the end of the operation. For example, if `connect()` is called, the socket will be put in an `ESTABLISHED` state as requested without checking for acknowledgment of communication from the other end, so offloading could be at the application level.
- **Desktop environment suspend/resume:** It is based on the fact that the suspend/restore action for a screen session or an X application is by far faster than the close/open operation.
- **High performance and computing issues:** It can be used for both load balancing of tasks over a cluster and the saving of cluster node states in case a crash occurs. Having a number of snapshots for application doesn't hurt anybody.
- **Duplication of processes:** It is similar to the remote `fork()` operation.
- **Snapshots for applications:** A series of application states can be saved and reversed back if necessary. It can be used both as a redo for the desired state of an application as well as for debugging purposes.
- **Save ability in applications that do not have this option:** An example of such an application could be games in which after reaching a certain level, the establishment of a checkpoint is the thing you need.
- **Migrate a forgotten application onto the screen:** If you have forgotten to include an application onto the screen and you are already there, CRIU can help with the migration process.
- **Debugging of applications that have hung:** For services that are stuck because of `git` and need a quick restart, a copy of the services can be used to restore. A dump process can also be used and through debugging, the cause of the problem can be found.
- **Application behavior analysis on a different machine:** For those applications that could behave differently from one machine to another, a snapshot of the application in question can be used and transferred into the other. Here, the debugging process can also be an option.
- **Dry running updates:** Before a system or kernel update on a system is done, its services and critical applications could be duplicated onto a virtual machine and after the system update and all the test cases pass, the real update can be done.
- **Fault-tolerant systems:** It can be used successfully for process duplication on other machines.

The next element is `irqbalance`, a distributed hardware interrupt system that is available across multiple processors and multiprocessor systems. It is, in fact, a daemon used to balance interrupts across multiple CPUs, and its purpose is to offer better performances as well as better IO operation balance on SMP systems. It has alternatives, such as `smp_affinity`, which could achieve maximum performance in theory, but lacks the same flexibility that `irqbalance` provides.

The `libvirt` toolkit can be used to connect with the virtualization capabilities available in the recent Linux kernel versions that have been licensed under the GNU Lesser General Public License. It offers support for a large number of packages, as follows:

- KVM/QEMU Linux supervisor
- Xen supervisor
- LXC Linux container system
- OpenVZ Linux container system
- Open Mode Linux a paravirtualized kernel
- Hypervisors that include VirtualBox, VMware ESX, GSX, Workstation and player, IBM PowerVM, Microsoft Hyper-V, Parallels, and Bhyve

Besides these packages, it also offers support for storage on a large variety of filesystems, such as IDE, SCSI or USB disks, FiberChannel, LVM, and iSCSI or NFS, as well as support for virtual networks. It is the building block for other higher-level applications and tools that focus on the virtualization of a node and it does this in a secure way. It also offers the possibility of a remote connection.

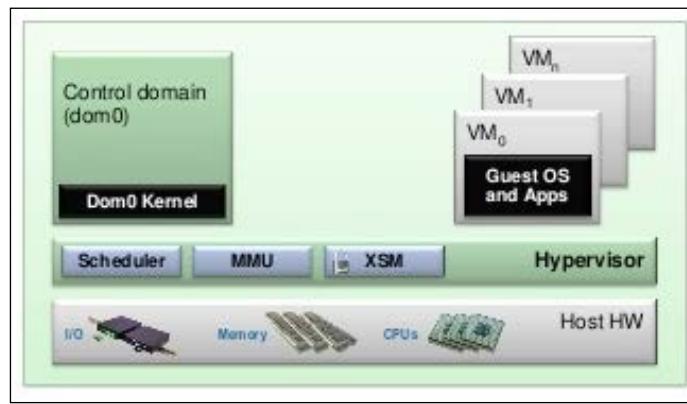


For more information about `libvirt`, take a look at its project goals and terminologies at <http://libvirt.org/goals.html>.

The next is `Open vSwitch`, a production-quality implementation of a multilayer virtual switch. This software component is licensed under Apache 2.0 and is designed to enable massive network automations through various programmatic extensions. The `Open vSwitch` package, also abbreviated as **OVS**, provides a two stack layer for hardware virtualizations and also supports a large number of the standards and protocols available in a computer network, such as sFlow, NetFlow, SPAN, CLI, RSPAN, 802.1ag, LACP, and so on.

Xen is a hypervisor with a microkernel design that provides services offering multiple computer operating systems to be executed on the same architecture. It was first developed at the Cambridge University in 2003, and was developed under GNU General Public License version 2. This piece of software runs on a more privileged state and is available for ARM, IA-32, and x86-64 instruction sets.

A hypervisor is a piece of software that is concerned with the CPU scheduling and memory management of various domains. It does this from the **domain 0 (dom0)**, which controls all the other unprivileged domains called **domU**; Xen boots from a bootloader and usually loads into the dom0 host domain, a paravirtualized operating system. A brief look at the Xen project architecture is available here:



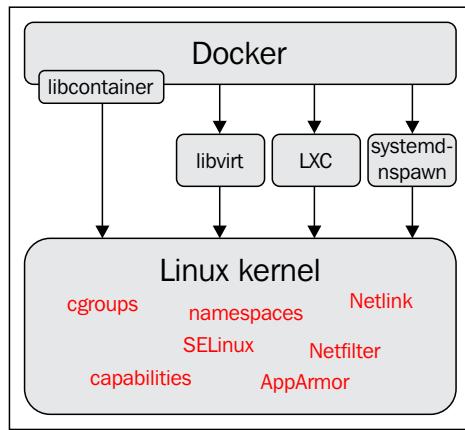
Linux Containers (LXC) is the next element available in the meta-virtualization layer. It is a well-known set of tools and libraries that offer virtualization at the operating system level by offering isolated containers on a Linux control host machine. It combines the functionalities of kernel **control groups (cgroups)** with the support for isolated namespaces to provide an isolated environment. It has received a fair amount of attention mostly due to Docker, which will be briefly mentioned a bit later. Also, it is considered a lightweight alternative to full machine virtualization.

Both of these options, containers and machine virtualization, have a fair amount of advantages and disadvantages. If the first option, containers offer low overheads by sharing certain components, and it may turn out that it does not have a good isolation. Machine virtualization is exactly the opposite of this and offers a great solution to isolation at the cost of a bigger overhead. These two solutions could also be seen as complementary, but this is only my personal view of the two. In reality, each of them has its particular set of advantages and disadvantages that could sometimes be uncomplementary as well.



More information about Linux containers is available at
<https://linuxcontainers.org/>.

The last component of the meta-virtualization layer that will be discussed is **Docker**, an open source piece of software that tries to automate the method of deploying applications inside Linux containers. It does this by offering an abstraction layer over LXC. Its architecture is better described in this image:



As you can see in the preceding diagram, this software package is able to use the resources of the operating system. Here, I am referring to the functionalities of the Linux kernel and have isolated other applications from the operating system. It can do this either through LXC or other alternatives, such as libvirt and systemd-nspawn, which are seen as indirect implementations. It can also do this directly through the libcontainer library, which has been around since the 0.9 version of Docker.

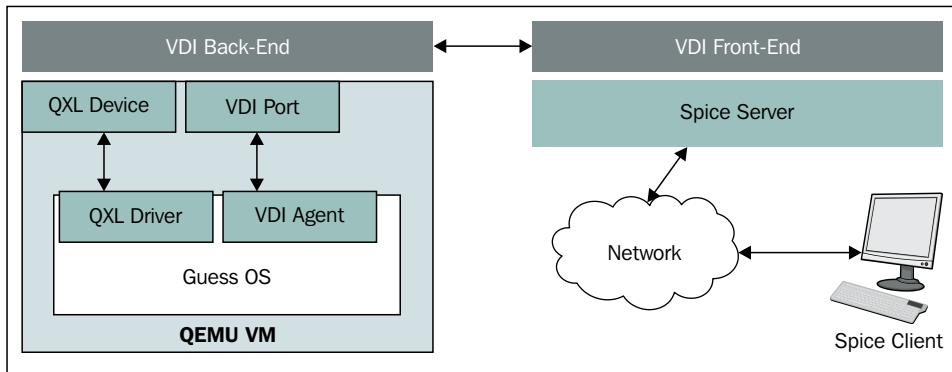
Docker is a great component if you want to obtain automation for distributed systems, such as large-scale web deployments, service-oriented architectures, continuous deployment systems, database clusters, private PaaS, and so on. More information about its use cases is available at <https://www.docker.com/resources/usecases/>. Make sure you take a look at this website; interesting information is often here.



More information about the Docker project is available on their website. Check out the **What is Docker?** section at <https://www.docker.com/whatisdocker/>.

After finishing with the meta-virtualization layer, I will move next to the meta-cloud-services layer that contains various elements. I will start with **Simple Protocol for Independent Computing Environments (Spice)**. This can be translated into a remote-display system for virtualized desktop devices.

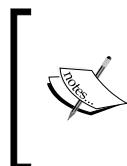
It initially started as a closed source software, and in two years it was decided to make it open source. It then became an open standard to interaction with devices, regardless of whether they are virtualized one not. It is built on a client-server architecture, making it able to deal with both physical and virtualized devices. The interaction between backend and frontend is realized through **VD-Interfaces (VDI)**, and as shown in the following diagram, its current focus is the remote access to QEMU/KVM virtual machines:



Next on the list is **oVirt**, a virtualization platform that offers a web interface. It is easy to use and helps in the management of virtual machines, virtualized networks, and storages. Its architecture consists of an oVirt Engine and multiple nodes. The engine is the component that comes equipped with a user-friendly interface to manage logical and physical resources. It also runs the virtual machines that could be either oVirt nodes, Fedora, or CentOS hosts. The only downfall of using oVirt is that it only offers support for a limited number of hosts, as follows:

- Fedora 20
- CentOS 6.6, 7.0
- Red Hat Enterprise Linux 6.6, 7.0
- Scientific Linux 6.6, 7.0

As a tool, it is really powerful. It offers integration with `libvirt` for **Virtual Desktops and Servers Manager (VDSM)** communications with virtual machines and also support for SPICE communication protocols that enable remote desktop sharing. It is a solution that was started and is mainly maintained by Red Hat. It is the base element of their **Red Hat Enterprise Virtualization (RHEV)**, but one thing is interesting and should be watched out for is that Red Hat now is not only a supporter of projects, such as oVirt and Aeolus, but has also been a platinum member of the OpenStack foundation since 2012.



For more information on projects, such as oVirt, Aeolus, and RHEV, the following links can be useful to you: http://www.redhat.com/promo/rhev3/?sc_cid=7016000000Ty5wAAC&offer_id=7016000000Ty5NAAS <http://www.aeolusproject.org/>, and <http://www.ovirt.org/Home>.



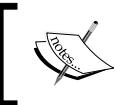
I will move on to a different component now. Here, I am referring to the open source implementation of the Lightweight Directory Access Protocol, simply called **OpenLDAP**. Although it has a somewhat controversial license called **OpenLDAP Public License**, which is similar in essence to the BSD license, it is not recorded at opensource.org, making it uncertified by **Open Source Initiative (OSI)**.

This software component comes as a suite of elements, as follows:

- A standalone LDAP daemon that plays the role of a server called **slapd**
- A number of libraries that implement the LDAP protocol
- Last but not the least, a series of tools and utilities that also have a couple of clients samples between them

There are also a number of additions that should be mentioned, such as `ldapc++` and libraries written in C++, `JLDAP` and the libraries written in Java; `LMDB`, a memory mapped database library; `Fortress`, a role-based identity management; `SDK`, also written in Java; and a `JDBC-LDAP Bridge` driver that is written in Java and called **JDBC-LDAP**.

Cyrus SASL is a generic client-server library implementation for **Simple Authentication and Security Layer (SASL)** authentication. It is a method used for adding authentication support for connection-based protocols. A connection-based protocol adds a command that identifies and authenticates a user to the requested server and if negotiation is required, an additional security layer is added between the protocol and the connection for security purposes. More information about SASL is available in the RFC 2222, available at <http://www.ietf.org/rfc/rfc2222.txt>.



For a more detailed description of Cyrus SASL, refer to <http://www.sendmail.org/~ca/email/cyrus/sysadmin.html>.



Qpid is a messaging tool developed by Apache, which understands **Advanced Message Queueing Protocol (AMQP)** and has support for various languages and platforms. AMQP is an open source protocol designed for high-performance messaging over a network in a reliable fashion. More information about AMQP is available at <http://www.amqp.org/specification/1.0/amqp-org-download>. Here, you can find more information about the protocol specifications as well as about the project in general.

Qpid projects push the development of AMQP ecosystems and this is done by offering message brokers and APIs that can be used in any developer application that intends to use AMQP messaging part of their product. To do this, the following can be done:

- Letting the source code open source.
- Making AMQP available for a large variety of computing environments and programming languages.
- Offering the necessary tools to simplify the development process of an application.
- Creating a messaging infrastructure to make sure that other services can integrate well with the AMQP network.
- Creating a messaging product that makes integration with AMQP trivial for any programming language or computing environment. Make sure that you take a look at Qpid Proton at <http://qpid.apache.org/proton/overview.html> for this.



More information about the preceding functionalities can be found at <http://qpid.apache.org/components/index.html#messaging-apis>.



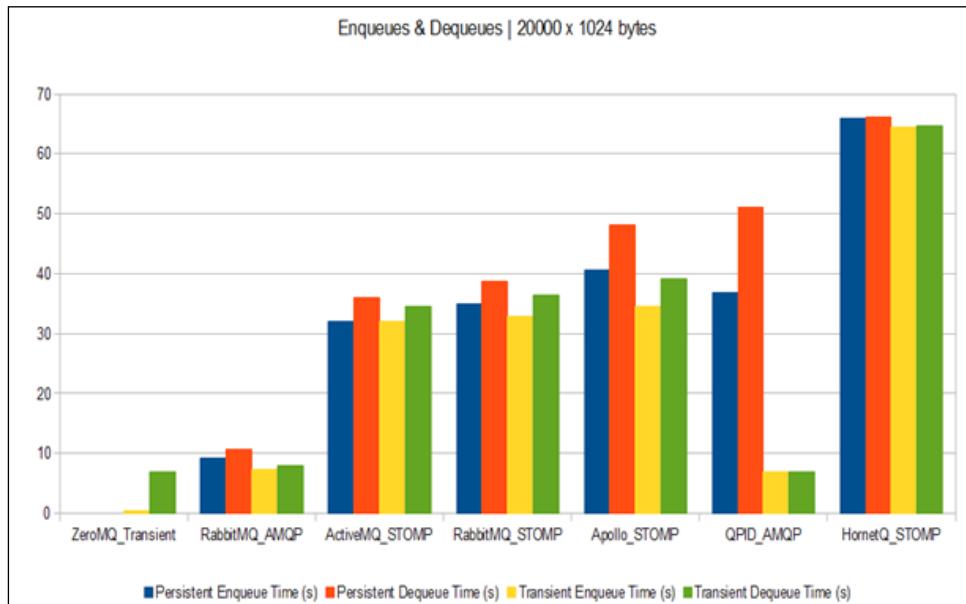
RabbitMQ is another message broker software component that implements AMQP, which is also available as open source. It has a number of components, as follows:

- The RabbitMQ exchange server
- Gateways for HTTP, **Streaming Text Oriented Message Protocol (STOMP)** and **Message Queue Telemetry Transport (MQTT)**

- AMQP client libraries for a variety of programming languages, most notably Java, Erlang, and .Net Framework
- A plugin platform for a number of custom components that also offer a collection of predefined one:
 - **Shovel:** It is a plugin that executes the copy/move operation for messages between brokers
 - **Management:** It enables the control and monitoring of brokers and clusters of brokers
 - **Federation:** It enables sharing at the exchange level of messages between brokers

[ You can find out more information regarding RabbitMQ by referring to the RabbitMQ documentation section at <http://www.rabbitmq.com/documentation.html>.]

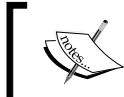
Comparing the two, Qpid and RabbitMQ, it can be concluded that RabbitMQ is better and also that it has a fantastic documentation. This makes it the first choice for the OpenStack Foundation as well as for readers interested in benchmarking information for more than these frameworks. It is also available at <http://blog.x-aeon.com/2013/04/10/a-quick-message-queue-benchmark-activemq-rabbitmq-hornetq-qpid-apollo/>. One such result is also available in this image for comparison purposes:



The next element is **puppet**, an open source configuration management system that allows IT infrastructure to have certain states defined and also enforce these states. By doing this, it offers a great automation system for system administrators. This project is developed by the Puppet Labs and was released under GNU General Public License until version 2.7.0. After this, it moved to the Apache License 2.0 and is now available in two flavors:

- **The open source puppet version:** It is mostly similar to the preceding tool and is capable of configuration management solutions that permit for definition and automation of states. It is available for both Linux and UNIX as well as Max OS X and Windows.
- **The puppet enterprise edition:** It is a commercial version that goes beyond the capabilities of the open source puppet and permits the automation of the configuration and management process.

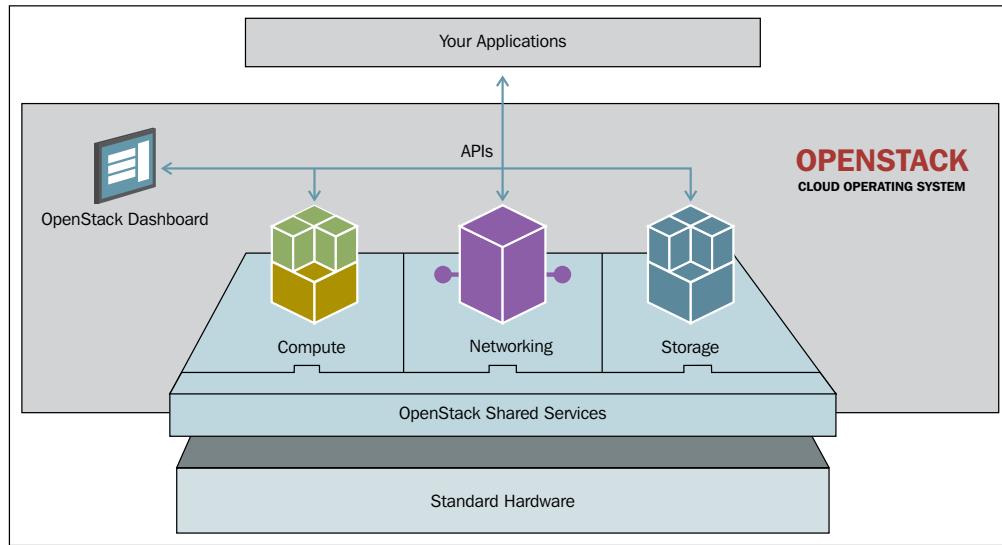
It is a tool that defines a declarative language for later use for system configuration. It can be applied directly on the system or even compiled as a catalogue and deployed on a target using a client-server paradigm, which is usually the REST API. Another component is an agent that enforces the resources available in the manifest. The resource abstraction is, of course, done through an abstraction layer that defines the configuration through higher lever terms that are very different from the operating system-specific commands.



If you visit <http://docs.puppetlabs.com/>, you will find more documentation related to Puppet and other Puppet Lab tools.



With all this in place, I believe it is time to present the main component of the meta-cloud-services layer, called **OpenStack**. It is a cloud operating system that is based on controlling a large number of components and together it offers pools of compute, storage, and networking resources. All of them are managed through a dashboard that is, of course, offered by another component and offers administrators control. It offers users the possibility of providing resources from the same web interface. Here is an image depicting the Open Source Cloud operating System, which is actually OpenStack:



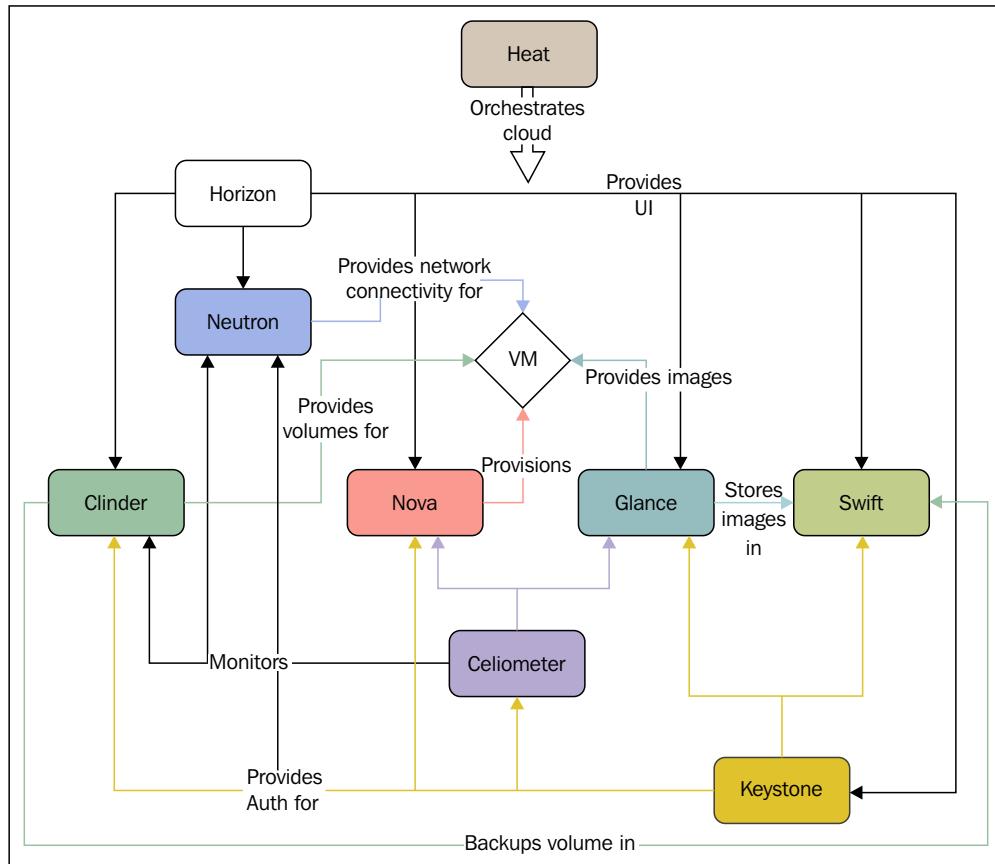
It is primarily used as an IaaS solution, its components are maintained by the OpenStack Foundation, and is available under Apache License version 2. In the Foundation, today, there are more than 200 companies that contribute to the source code and general development and maintenance of the software. At the heart of it, all are staying its components Also, each component has a Python module used for simple interaction and automation possibilities:

- **Compute (Nova):** It is used for the hosting and management of cloud computing systems. It manages the life cycles of the compute instances of an environment. It is responsible for the spawning, decommissioning, and scheduling of various virtual machines on demand. With regard to hypervisors, KVM is the preferred option but other options such as Xen and VMware are also viable.
- **Object Storage (Swift):** It is used for storage and data structure retrieval via RESTful and the HTTP API. It is a scalable and fault-tolerant system that permits data replication with objects and files available on multiple disk drives. It is developed mainly by an object storage software company called **SwiftStack**.

- **Block Storage (Cinder):** It provides persistent block storage for OpenStack instances. It manages the creation and attach and detach actions for block devices. In a cloud, a user manages its own devices, so a vast majority of storage platforms and scenarios should be supported. For this purpose, it offers a pluggable architecture that facilitates the process.
- **Networking (Neutron):** It is the component responsible for network-related services, also known as **Network Connectivity as a Service**. It provides an API for network management and also makes sure that certain limitations are prevented. It also has an architecture based on pluggable modules to ensure that as many networking vendors and technologies as possible are supported.
- **Dashboard (Horizon):** It provides web-based administrators and user graphical interfaces for interaction with the other resources made available by all the other components. It is also designed keeping extensibility in mind because it is able to interact with other components responsible for monitoring and billing as well as with additional management tools. It also offers the possibility of rebranding according to the needs of commercial vendors.
- **Identity Service (Keystone):** It is an authentication and authorization service. It offers support for multiple forms of authentication and also existing backend directory services such as LDAP. It provides a catalogue for users and the resources they can access.
- **Image Service (Glance):** It is used for the discovery, storage, registration, and retrieval of images of virtual machines. A number of already stored images can be used as templates. OpenStack also provides an operating system image for testing purposes. Glance is the only module capable of adding, deleting, duplicating, and sharing OpenStack images between various servers and virtual machines. All the other modules interact with the images using the available APIs of Glance.
- **Telemetry (Ceilometer):** It is a module that provides billing, benchmarking, and statistical results across all current and future components of OpenStack with the help of numerous counters that permit extensibility. This makes it a very scalable module.

- **Orchestrator (Heat):** It is a service that manages multiple composite cloud applications with the help of various template formats, such as Heat Orchestration Templates (HOT) or AWS CloudFormation. The communication is done both on a CloudFormation compatible Query API and an Open Stack REST API.
- **Database (Trove):** It provides Cloud Database as service functionalities that are both reliable and scalable. It uses relational and nonrelational database engines.
- **Bare Metal Provisioning (Ironic):** It is a components that provides virtual machine support instead of bare metal machines support. It started as a fork of the Nova Baremetal driver and grew to become the best solution for a bare-metal hypervisor. It also offers a set of plugins for interaction with various bare-metal hypervisors. It is used by default with PXE and IPMI, but of course, with the help of the available plugins it can offer extended support for various vendor-specific functionalities.
- **Multiple Tenant Cloud Messaging (Zaqar):** It is, as the name suggests, a multitenant cloud messaging service for the web developers who are interested in **Software as a Service (SaaS)**. It can be used by them to send messages between various components by using a number of communication patterns. However, it can also be used with other components for surfacing events to end users as well as communication in the over-cloud layer. Its former name was **Marconi** and it also provides the possibility of scalable and secure messaging.
- **Elastic Map Reduce (Sahara):** It is a module that tries to automate the method of providing the functionalities of Hadoop clusters. It only requires the defines for various fields, such as Hadoop versions, various topology nodes, hardware details, and so on. After this, in a few minutes, a Hadoop cluster is deployed and ready for interaction. It also offers the possibility of various configurations after deployment.

Having mentioned all this, maybe you would not mind if a conceptual architecture is presented in the following image to present to you with ways in which the above preceding components are interacted with. To automate the deployment of such an environment in a production environment, automation tools, such as the previously mentioned Puppet tool, can be used. Take a look at this diagram:



Now, let's move on and see how such a system can be deployed using the functionalities of the Yocto Project. For this activity to start, all the required metadata layers should be put together. Besides the already available Poky repository, other ones are also required and they are defined in the layer index on OpenEmbedded's website because this time, the README file is incomplete:

```

git clone -b dizzy git://git.openembedded.org/meta-openembedded
git clone -b dizzy git://git.yoctoproject.org/meta-virtualization
git clone -b icehouse git://git.yoctoproject.org/meta-cloud-
services
source oe-init-build-env ../build-controller

```

After the appropriate controller build is created, it needs to be configured. Inside the `conf/layer.conf` file, add the corresponding machine configuration, such as `qemux86-64`, and inside the `conf/bblayers.conf` file, the `BBLAYERS` variable should be defined accordingly. There are extra metadata layers, besides the ones that are already available. The ones that should be defined in this variable are:

- `meta-cloud-services`
- `meta-cloud-services/meta-openstack-controller-deploy`
- `meta-cloud-services/meta-openstack`
- `meta-cloud-services/meta-openstack-qemu`
- `meta-openembedded/meta-oe`
- `meta-openembedded/meta-networking`
- `meta-openembedded/meta-python`
- `meta-openembedded/meta-filesystem`
- `meta-openembedded/meta-webserver`
- `meta-openembedded/meta-ruby`

After the configuration is done using the `bitbake openstack-image-controller` command, the controller image is built. The controller can be started using the `rungemu qemux86-64 openstack-image-controller kvm nographic qemuparams="-m 4096"` command. After finishing this activity, the deployment of the compute can be started in this way:

```
source oe-init-build-env ..../build-compute
```

With the new build directory created and also since most of the work of the build process has already been done with the controller, build directories such as `downloads` and `sstate-cache`, can be shared between them. This information should be indicated through `DL_DIR` and `SSTATE_DIR`. The difference between the two `conf/bblayers.conf` files is that the second one for the `build-compute` build directory replaces `meta-cloud-services/meta-openstack-controller-deploy` with `meta-cloud-services/meta-openstack-compute-deploy`.

This time the build is done with `bitbake openstack-image-compute` and should be finished faster. Having completed the build, the compute node can also be booted using the `rungemu qemux86-64 openstack-image-compute kvm nographic qemuparams="-m 4096 -smp 4"` command. This step implies the image loading for OpenStack Cirros as follows:

```
wget download.cirros-cloud.net/0.3.2/cirros-0.3.2-x86_64-disk.img  
scp cirros-0.3.2-x86_64-disk.img root@<compute_ip_address>:~
```

```
ssh root@<compute_ip_address>
./etc/nova/openrc
glance image-create --name "TestImage" --is-public true --container-
format bare --disk-format qcow2 --file /home/root/cirros-0.3.2-x86_64-
disk.img
```

Having done all of this, the user is free to access the Horizon web browser using `http://<compute_ip_address>:8080/`. The login information is admin and the password is password. Here, you can play and create new instances, interact with them, and, in general, do whatever crosses your mind. Do not worry if you've done something wrong to an instance; you can delete it and start again.

The last element from the `meta-cloud-services` layer is the **Tempest integration test suite** for OpenStack. It is represented through a set of tests that are executed on the OpenStack trunk to make sure everything is working as it should. It is very useful for any OpenStack deployments.



More information about Tempest is available at
<https://github.com/openstack/tempest>.



Summary

In this chapter, you were not only presented with information about a number of virtualization concepts, such as NFV, SDN, VNF, and so on, but also a number of open source components that contribute to everyday virtualization solutions. I offered you examples and even a small exercise to make sure that the information remains with you even after reading this book. I hope I made some of you curious about certain things. I also hope that some of you documented on projects that were not presented here, such as the **OpenDaylight (ODL)** initiative, that has only been mentioned in an image as an implementation suggestion. If this is the case, I can say I fulfilled my goal. If not, maybe this summary will make you go through the previous pages again.

In the next chapter, we will visit a new and real carrier graded one. It will be the last chapter of this book and I will conclude it with a topic that is very important to me personally. I will discuss the Yocto shy initiative called **meta-cgl** and its purpose. I will present the various specifications and changes for the **Carrier Graded Linux (CGL)**, and the requirements of **Linux Standard Base (LSB)**. I hope you enjoy reading it as much as I have enjoyed writing it.

13

CGL and LSB

In this chapter, you will be presented with information about the last topic of the book, the **Carrier Grade Linux (CGL)** and **Linux Standard Base (LSB)** initiative and of course, a parallel with what there is integrated and supported related to those two standards into the Yocto Project. This will also be mentioned here and you will not only be able to find a little bit about these standards and their specifications, but also about the level of support that Yocto offers for them. I will also present some of the initiatives adjacent to CGL, such as **Automotive Grade Linux** and **Carrier Grade Virtualization**. They also constitute viable solutions that are available in a wide palette of applications.

In any Linux environment today, there is necessity for a common language for available Linux distributions. This common language would have not been achieved without defining actual specifications. A part of these specifications is also represented by the carrier grade alternative. It coexists with other specifications that are already presented in this book or in other similar books. Taking a look at the available specifications and standardizations only shows us how much the Linux ecosystem has evolved over time.

The latest report published by the guys working at the Linux Foundation shows how the development of the Linux kernel is actually done nowadays, what it's like to work on it, who is sponsoring it, what changes are being made to it, and how fast things are moving. The report is available at <https://www.linuxfoundation.org/publications/linux-foundation/who-writes-linux-2015>.

As depicted in the report, less than 20 percent of development on the kernel is done by individual developers. Most of the development is realized by companies, such as Intel, Red Hat, Linaro, Samsung, and others. This means that over 80 percent of the developers working at Linux kernel development are paid for their job. The fact that Linaro and Samsung are some of the companies with the most number of commits, only presents a favorable perception of the ARM processors in general, and Android in particular.

Another interesting piece of information is that more than half of the Linux kernel developers are at their first commit. This means that a really small number of developers are doing the vast majority of work. This dysfunction in the development of the Linux kernel process is being tried to be reduced by the Linux Foundation by offering various programs for students to make them more involved in the development process. Whether this is a success, only time will tell, but it is my opinion that they are doing the right thing and are moving in the right direction.

All of this information has been explained with regard to the Linux kernel, but parts of it are applicable for other open source components. The thing that I want to emphasize here is that the ARM support in Linux is much more mature than in architectures such as PowerPC or MIPS. This has started to not only be obvious, but is also an indication of the approach that the Intel x86 stage has taken. Until now, this approach was simply not disturbed by anyone.

Linux Standard Base

LSB appeared to lower the costs of support offered by Linux platforms by reducing the differences between various available Linux distributions. It also helps with costs for porting applications. Every time a developer writes an application, they need to make sure that the source code produced on one Linux distribution will also be able to be executed on other distributions as well. They would also like to make sure that this remains possible over the years.

The LSB workgroup is a Linux Foundation project that tries to address these exact problems. For this purpose, LSB workgroup started working on a standard that could describe a set of APIs that a Linux distribution should support. With the standards defined, the workgroup also moved a few steps further and developed a set of tools and tests to measure the support levels. With this done, they were able to define certain sets of compliance and also detect the certain differences between various distributions.

The LSB was the first effort to be made in this direction by the Linux Foundation and became an umbrella for all the workgroups that have tried to provide standardization to various areas of the Linux platform. All these workgroups have the same roadmap and they deliver their corresponding set of specifications, software components, such as conformance tests, developments tools, and other available samples and implementations.

Every software component developed by one of the workgroups that is available inside the Linux Standard Base is defined as a `1sb` module. All of these modules have a common format to facilitate easier integration between them. There are modules that are required and optional. The required ones are the ones that meet the acceptance criteria for LSB. The optional ones are still a work in progress and are, at the moment of specifications defining, not written in the acceptance criteria, but will be included in future versions of the LSB standard.

There are, of course, workgroups that do not produce `1sb` modules. They have not worked on the standard either but instead, they have integrated various patches in projects, such as the Linux kernel or other packages and even documentation. These are not the workgroups that this section is referring to. This section only takes LSB-related workgroups into account.

From time to time, whenever a new specification document is released, a testing kit is also made available to vendors to test the kit's compliance to a particular version. The vendors could test their product compliance, which can be in the form of an application or a Linux distribution. The result of the testing kit is a certification that indicates that their product is LSB certified. For an application we, of course, have an **LSB Application Testkit**. There is also a similar one for a Linux distribution as well as others that are available for a variety of distributions.

For vendors who are interested in optional modules, these are not only available to help vendors prepare their future LSB compliance certification, but also to expose them to optional modules in order to get more vocal reviews and contributions from them. Also, the vendor's vote is related to the existence of these modules in future LSB specification documentations whose release is also important. The vendors could establish whether one optional module is eligible for future inclusions or not.

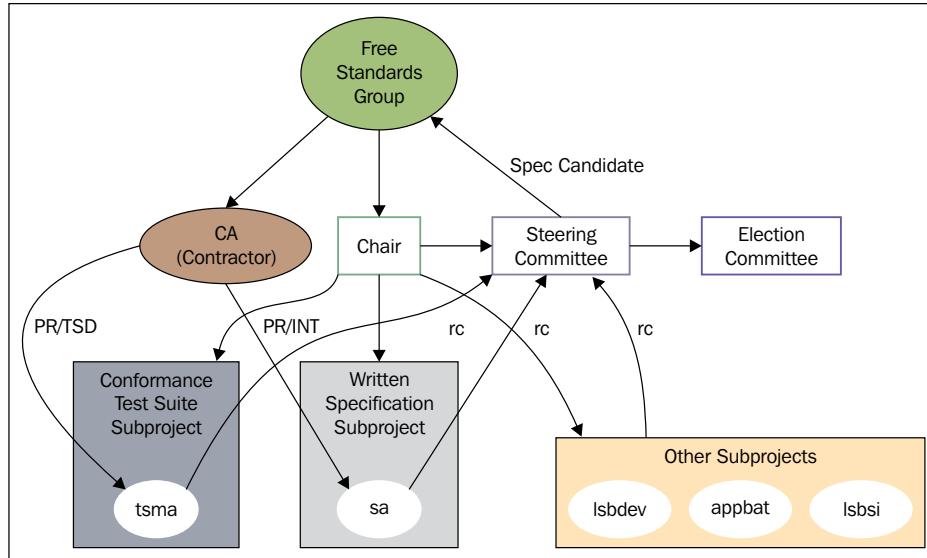
The LSB workgroup is governed by the Steering Committee and is led by a Chairperson who is elected. These two entities represent the interests of the workgroup. The workgroup operates on a rough consensus model. This indicates the solution of the group regarding a particular problem, that is, a solution that is determined by the elected Chairperson. If the contributor does not consider their decision and does not meet the criteria required to reach a rough consensus, then the Steering committee is appealed.

All business that is specific to the LSB workgroup is carried out inside an open forum. It can include a mailing list, conference, wiki page, or even a face-to-face meeting; the activities are not closed for members of workgroups. Also, membership is not restricted and decisions are clearly documented because there is always a possibility of having a further discussion on a particular subject at a later time.

There are clearly defined roles in workgroups:

- **Contributor:** This refers to actively involved individuals. They always have a list with them available for the Chairperson, but any individual may request inclusion to the Contributors list.
- **Chairperson:** This refers to the representative Project leader. A person is elected to this position by Contributors and approved by the Steering Committee and the Linux Foundation board. Once elected, they are able to hold this position for two years. There is no limit to the number of times someone can be elected. Removal from this position can occur in case of a lack of confidence on behalf of the Steering Committee or the Linux Foundation board. After the position is vacant, a new election is carried out. During the vacancy period, the Steering Committee will assign an acting Chairperson.
- **Election Committee:** This refers to a committee of Contributors that are established by the Steering Committee for Chairperson election. It is responsible for selecting candidates for the position of Chairperson within at least 30 days before the Chairperson's term expires or 10 days after the Chairperson's position is vacant. It is responsible for conducting elections, which is done through electronic ballots. There is only one vote accepted from an individual; the votes are secret and only done by eligible members. The voting period is one week, and then the results are presented to the Steering Committee, which approves the votes and declares the winner.
- **Steering Committee:** It consists of representative workgroup stakeholders. They may be distribution vendors, OEMs, ISVs, upstream developers, and the Chairpersons of the LSB sub workgroups that come under the LSB charter. The committee is appointed by the Chairperson and depending on their involvement in workgroup activities, they can keep the position indefinitely. One member can be removed from the Steering Committee by three entities: the Chairperson, the other Steering Committee members, or by the Linux Foundation board.

Here is an image depicting a more detailed structure of the LSB workgroup:



The LSB is a fairly complex structure, as depicted in the preceding image, so more roles can be defined in a workgroup if necessary. The main focus of the workgroup remains its mission; for this to be achievable, new workgroups need to be promoted and nurtured. They require a certain level of independence, but also be accountable for the activities done in the LSB Chairperson. This mainly involves making sure that certain deadlines are met and that the project sticks to its roadmap.

The first step in the interaction process with the LSB deliverables should be establishing the exact LSB requirements that need to be met by a target system. The specifications are available as two components: architecture-dependent and architecture-independent, or as it is also called, a generic component. The architecture-dependent components contain three modules:

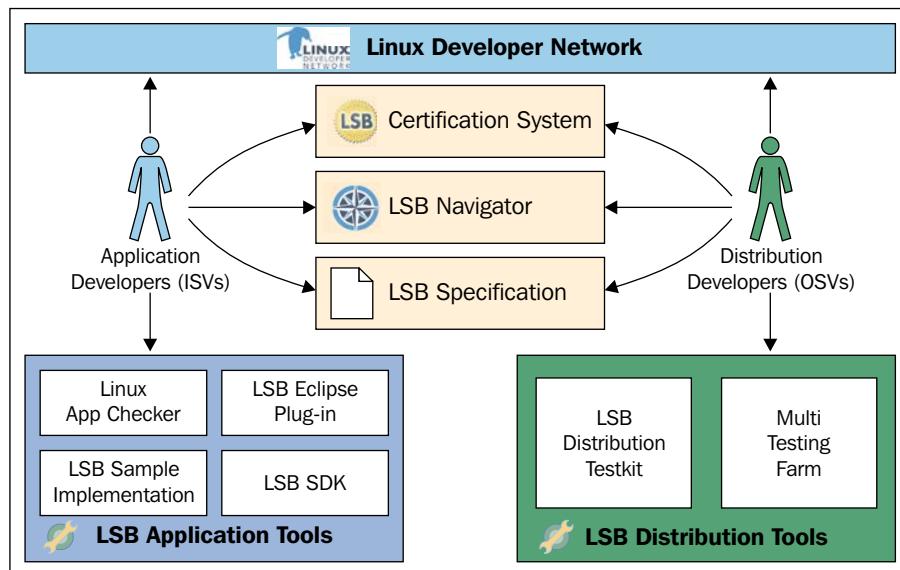
- Core
- C++
- Desktop

Architecture independent components contain five modules:

- Core
- C++
- Desktop
- Printing
- Languages

There is, of course, another structure used to order them. Here, I am referring to the fact that some of these are mandatory and others are in a state of trial and testing. The first category is in order to have a distribution that is compliant with LSB standards, while the second category is not a strict requirement for having a compliant distribution that could represent future candidates for the next few versions of LSB.

The following image represents the key deliverable components of LSB. I hope it guides you through the components of this project as well as gathers the information that you need for future interaction with the various components of the LSB workgroup.



Depending on the interest of users, they can chose to either interact with the distribution development or the development of the components of an application. As clearly depicted in the preceding image, each of the two roads has its tools for the job. Before starting a job, make sure that you take a look at the website of the LSB Navigator and gather the required information. For users who are interested in a demonstration of LSB navigator, there is one available in the following link that also involves the interaction of Yocto. Make sure that you check it out and interact with it to get an idea of how it works.



The LSB Navigator can be accessed at <http://www.linuxbase.org/navigator/commons/welcome.php>.



Let's assume that the interaction is already done and you are now interested in collaborating with this project. Of course, there are multiple methods to do this. Whether you are a developer or a software vendor, your feedback is always helpful for any project. Also, for developers who would like to contribute with code, there are multiple components and tools that could benefit from your help. That is not all. There are a lot of testing frameworks and testing infrastructures that always require improvements, so someone can contribute not only with code but also bug fixing and development or the testing of tools. Also, remember that your feedback is always appreciated.

Before moving to the next section, I want to introduce one more thing. As depicted in the previous diagram, any activity that is executed by a developer, with regard to the components of the LSB workgroup, should be done after the LSB specifications are inspected and the appropriate version is selected. For example, in the CGL Specifications, there is an explicit requirement of at least LSB 3.0, as well as the required modules, that are indicated in the same requirement description. For developers who want more information about the required specification and its components, refer to <http://refspecs.linuxfoundation.org/lsb.shtml>. Make sure that you also inspect the progress made on the newly available LSB 5 specifications, which passed the beta stage and, at the moment, is in its RC1 state. More information about this is available at <https://www.linuxfoundation.org/collaborate/workgroups/lsb/lsb-50-rc1>.



More information about the LSB is available at <http://www.linuxfoundation.org/collaborate/workgroups/lsb>.



Carrier grade options

Multiple options will be discussed in this section, and we'll start by defining the term *carrier grade*. This seems like the perfect start. So, what does this term mean in a telecommunications environment? It refers to a system, software, and even hardware components that are really reliable. Here, I am not referring only to the five-nines or six-nines that CGL provides because not all industries and scenarios require this kind of reliability. We are only going to refer to something that can be defined as reliable in the scope of a project. For a system, software, or hardware component to be defined as carrier grade, it should also prove itself as well tested along with all sorts of functionalities, such as high availability, fault tolerance, and so on.

These five-nines and six-nines refer to the fact that a product is available 99.999 or 99.9999 percent of the time. This translates per year in a downtime of around 5 minutes for five-nines and 30 seconds for six-nines requirements. Having explained this, I will move on and present the available options of carrier grade.

Carrier Grade Linux

It is the first and oldest option available. It appeared as a necessity for the telecommunication industry in order to define a set of specifications, which in turn defined a set of standards for Linux-based operating systems. After implementations, this would make the system carrier grade capable.

The motivation behind the CGL is to present an open architecture as a possible solution or an alternative to the already available proprietary and closed source available solutions that were already available in telecommunication systems. The open architecture alternative is the best not only because it avoids a monolithically form, is not hard to maintain, scale, and develop, but also it offers the advantage of speed. It is faster and cheaper to have a system that is decoupled and makes its components accessible to a larger number of software or hardware engineers. All of these components would be able to serve the same purpose in the end.

The workgroup was initially started by the **Open Source Development Lab (OSDL)**, which after its merger with Free Standards Group formed The Linux Foundation. Now all the work moved there together with the workgroup. The latest available release for CGL is 5.0 and it includes registered Linux distributions, such as Wind River, MontaVista, and Red Flag.

The OSDL CGL workgroup has three categories of applications that CGL could fit into:

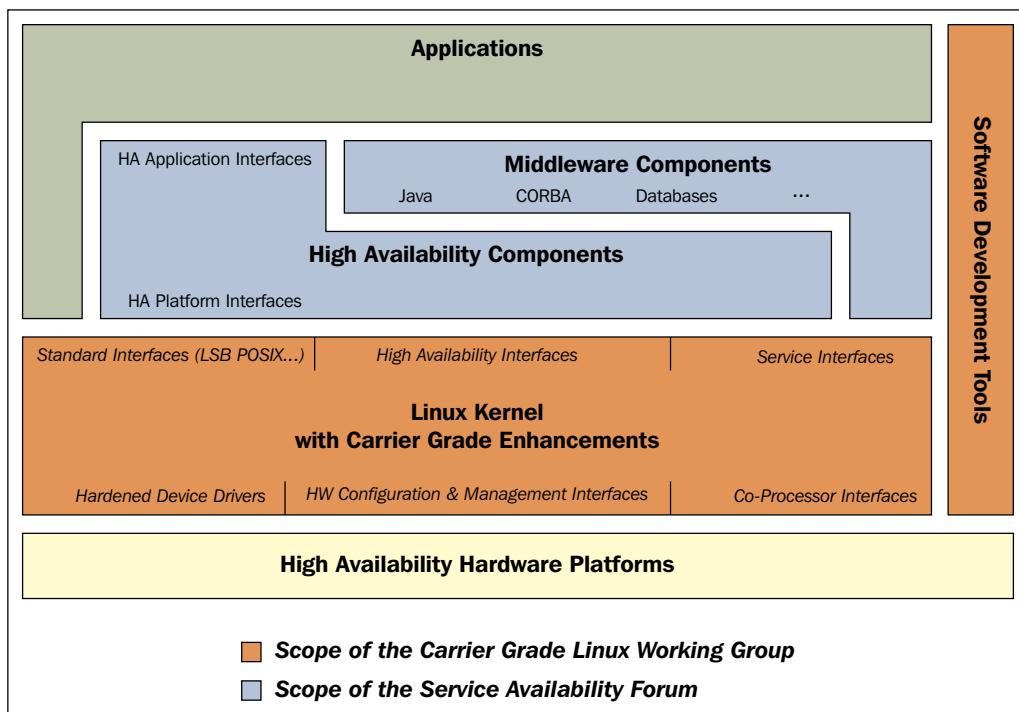
- **Signalling server applications:** This includes products that provide control services for calls and services, such as routing, session control, and status. These products usually handle a large number of connections, around 10000 or 100000 simultaneous ones, and also because that they have real-time requirements that require obtaining results from processes under a millisecond.
- **Gateway applications:** These provide the bridging of technology and administrative domains. Besides the characteristics that have been mentioned already, these handle a large number of connections in a real-time environment over a not very large number of interfaces. These are also required to not lose frames or packages in the communication process.
- **Management applications:** These usually provide billing operations, network management, and other traditional services. They does not have the same strong requirements for real-time operations, but instead, concentrate on fast database operations and other communication-oriented requests.

To make sure that it is able to satisfy the preceding categories, the CGL workgroup focuses on two main activities. The first one involves communicating with all the preceding categories, the identification of their requirements, and the writing specifications that should be implemented by distribution vendors. The second one involves gathering and helping projects that meet the requirements defined in the specifications. As a conclusion to what I mentioned previously, CGL tries to represent not only the telecommunication industry representatives and Linux distributions, but also end users and service providers; it also provides carrier grade options for each one of these categories.

Each distribution vendor who wants to get the CGL certification offers its implementation as a template. It is filled with versions of packages, names, and other extra information. However, it does this without disclosing too much information about the implementation process; these packages have the possibility of being proprietary software. Also, the disclosed information is owned and maintained by the vendor. The CGL workgroup only displays the link offered by the vendor.

The specification document is now at version 5.0 and contains both requirements that are, in fact, mandatory for applications or optional and are related to the implementations made in the Linux distribution for a carrier grade certification. The mandatory ones are described by the P1 priority level and the optional ones are marked as P2. The other elements are related to the gap aspect that represents a functionality, which is not implemented since an open source implementation is not available for it. The requirements are presented in the specification document to motivate distribution developers contribute to them.

As depicted in the following image and as emphasized in the information contained in the specification document, the CGL system should provide a large number of functionalities:



Since the requirement for number of functionalities is big, the workgroup decided to group them into various categories as follows:

- **Availability:** It is relevant for single node availability and recovery.
- **Clustering:** It describes components that are useful in building a cluster from individual systems. The key target behind this is the high availability of the system and load balancing that could also bring some performance improvements.

- **Serviceability:** It covers the maintenance and servicing features of the system.
- **Performance:** It describes features, such as real-time requirements and others, that could help the system attain better performance.
- **Standards:** These are provided as references to various APIs, standards, and specifications.
- **Hardware:** It presents various hardware-specific support that is necessary for a carrier grade operating system. Much of it comes from hardware vendors who are themselves involved in this process and the requirements from this section has been highly diminished in the latest CGL specification release.
- **Security:** It represents the relevant features needed to build a secure system.

 For more information on CGL requirements, refer to https://www.linuxfoundation.org/sites/main/files/CGL_5.0_Specification.pdf. You can also refer to the CGL workgroup at <https://www.linuxfoundation.org/collaborate/workgroups/cgl>.

Automotive Grade Linux

Automotive Grade Linux is also a Linux Foundation workgroup. It is newly formed and tries to offer an open source solution that has automotive applications. Its primary focus is the In-Vehicle-Infotainment sector, but it includes telematics systems and instrument clusters. Its efforts are based on open source components that are already available. These are suitable for its purposes and try to also enable rapid development, which is much needed in this industry.

The goals of the workgroup are:

- A transparent, collaborative, and open environment for involved elements.
- A Linux operating system stack that is focused on automotives and uses the open source community represented by exponents, such as developers, academic components, and companies as back support.
- A collective voice for interaction in the open source community released this time in the reverse form, from the AGL to the community.
- An embedded Linux distribution used for fast prototyping.

By using projects, such as Tizen, as the reference distribution and having projects, such as Jaguar, Nissan, Land Rover, or Toyoto, this project is interesting enough to be followed closely. It has just been developed but has potential for improvements. For those of you interested in it, refer to <https://www.linuxfoundation.org/collaborate/workgroups/automotive-grade-linux>. The project's wiki page is an interesting resource and can be consulted at <https://wiki.automotivelinux.org/>.

Carrier Grade Virtualization

The recent development of CGL made virtualization an interesting option for the carrier grade field because it involved a reduction in costs as well as transparency in leveraging multicore equipment that runs single-core designed applications. Virtualization options also needed to meet the same expectations as the other carrier grade systems.

Carrier Grade Virtualization has tried to become a vital component to be integrated in carrier grade platforms that are already available. This is done to preserve the attributes and performance of the system. It also tries to extend the appliance target and permits **Original Equipment Manufacturer (OEM)** to derive the benefits from the same support as the CGL. These benefits are in the form of well established targets.

Virtualization's application is more widespread, which can be seen ranging from the x86 architecture to ARM and DSP-based processors as well as a variety of domains. The examination of virtualization from a carrier grade point of view is the focus of this solution because, in this way, you can get a clearer perspective of the areas that require improvements. In this way, these can be identified and enhancements can also be applied as required. Unfortunately, this initiative has not been as exposed as some other ones, but is still a very good source of documentation and is available from virtualLogix at http://www.linuxpundit.com/documents/CGV_WP_Final_FN.pdf. I hope you enjoy its content.

Specific support for the Yocto Project

In the Poky reference system, support is provided for the development of LSB and LSB compatible applications. Inside Poky, there is a special `poky-1sb.conf` distribution policy configuration that is defined in case a distribution is interested in developing applications that are LSB-compliant. This holds true when generating a Linux distribution that is LSB-compliant or at least prepares to take the LSB certification. The build steps required for a Linux distribution that prepares for an LSB certification will be presented here. In case you are interested in developing LSB-compliant applications, the process is simpler and will also be briefly presented here; however, it is in contrast to the former.

The first step is simple: it only requires cloning the `poky` repository and the `meta-qt3` dependency layer because of the requirements of the LSB modules:

```
git clone git://git.yoctoproject.org/poky.git
git clone git://git.yoctoproject.org/meta-qt3
```

Next, the build directory needs to be created:

```
source oe-init-build-env -b ..../build_1sb
```

Inside the `conf/bblayers.conf` file, only the `meta-qt3` layer needs to be added. Inside the `conf/local.conf` file, the corresponding machine should be selected. I would suggest a capable platform, but using an emulated architecture, such as `qemuppc`, ought to be enough for such a demo if enough CPU power and memory is offered to it. Also, make sure that you change the `DISTRO` variable to `poky-1sb`. Having all these in place, the build process can start. The command necessary for this is:

```
bitbake core-image-1sb
```

After the resulting binaries are generated and booted on the selected machine, the user is able to either run all the tests using the `LSB_Test.sh` script, which also sets the LSB test framework environment, or run specific test suites:

```
/usr/bin/LSB_Test.sh
```

You can also use the following command:

```
cd /opt/lsb/test/manager/utils  
. ./dist-checker.pl -update  
. ./dist-checker.pl -D -s 'LSB 4.1' <test_suite>
```

If various tests are not passing, the system needs to be reconfigured to ensure the required compatibility level. Inside `meta/recipes-extended/images`, besides the `core-image-lsb.bb` recipes, there are also two similar recipes:

- `core-image-lsb-sdk.bb`: It includes a `meta-toolchain` and the necessary libraries and development headers that are needed to generate an SDK for application development
- `core-image-lsb-dev.bb`: It is suitable for development work on targets since it includes `dev-pkgs`, which exposes the necessary headers and libraries for image-specific packages

Inside the Yocto Project, is a layer defined as `meta-cgl`, which intends to be the stepping stone for the CGL initiative. It aggregates all the available and required packages defined by the CGL workgroup. This layer's format tries to set the stage for the next implementations that will be made to support CGL on various machines.

Inside the `meta-cgl` layer, there are two subdirectories:

- `meta-cgl-common`: It is the focus place of the activity and the subdirectory that offers support for machines available inside Poky, such as `qemuarm`, `qemuppc`, and so on.
- `meta-cgl-fsl-ppc`: It is a subdirectory that defines BSP-specific support. Such layers should be made available if the support for other machines is required.

As I've already mentioned, the `meta-cgl` layer is responsible for the CGL support. As mentioned previously, one of the requirements of CGL is to have LSB support and this support is available inside Poky. It is integrated inside this layer as a specific requirement. Another recommendation for the `meta-cgl` layer is to group all the available packages into package groups that define various categories. The available package groups are very generic, but all the available ones are integrated in a core one called `packagegroup-cgl.bb`.

The layer also exposes a CGL-compliant operating system image. This image tries to include various CGL-specific requirements for starters, and intends to grow by including all the requirements defined in the CGL specification document. Besides the resultant Linux operating system that will be compliant with the CGL requirements and is ready for the CGL certification, the layer also tries to define a CGL-specific testing framework. The task may seem similar to the one required for the LSB checking compliance, but I assure you it is not. It not only requires a CGL-specific language definition that has to be made according to the defined specifications, but also a number of tests definitions that should be in sync with what the language defines. Also, there are requirements that could be met with one package or the functionality of a package and these things should be gathered together and combined. There are various other scenarios that can be interpreted and answered correctly; this is a condition that makes the testing of CGL a hard task to accomplish.

Inside the `meta-cgl` layer, there are recipes for the following packages:

- `cluster-glue`
- `cluster-resource-agents`
- `corosync`
- `heartbeat`
- `lksctp-tools`
- `monit`
- `ocfs2-tools`
- `openais`
- `pacemaker`
- `openipmi`

Besides these recipes, there are also other ones that are necessary for various CGL requirements. The fact that the `meta-cgl` initiative is shown in the support it offers as described in the previous sections. It is not complete but it will be in time. It will also contain these packages:

- `evlog`
- `mipv6-daemon-umip`
- `makedumpfile`

All of these are necessary to offer a Linux-based operating system that has LSB support and CGL compliance. This will be carried out in time, and maybe by the time this book reaches your hands, the layer will be in its final format and be the standard for CGL compliance.

I will now start to explain a couple of packages that you might come across in the CGL environment. I will first start with the Heartbeat daemon, which provides communication and membership for cluster services. Having it in place will enable clients to determine the present state of the processes available on other machines and establish communication with them.

To make sure that the Heartbeat daemon is useful, it needs to be put together with a **Cluster Resource Manager (CRM)**, which is the component responsible for starting and stopping various services to obtain a highly available Linux system. This CRM was called **Pacemaker** and it was unable to detect resource-level fails and was only able to interact with two nodes. In time, it evolved, and it now has better support and additional user interfaces available. Some of these services are as follows:

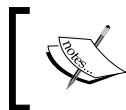
- **crm shell**: It is a command-line interface realized by Dejan Muhamedagic to hide the XML configuration and help with interactions.
- **The high availability web console**: It is an AJAX frontend
- **Heartbeat GUI**: It is an advanced XML editor that offers a lot of relevant information
- **Linux Cluster Management Console (LCMC)**: It started as **DRBD-Management Console (DRBD-MC)** and is a Java platform that is used for the management purposes of Pacemaker.

Pacemaker accepts three types of resource agents (a resource agent represents a standard interface between the cluster resources). The Resource Agents is a project that is also managed by Linux-HA. It is available and maintained by the guys at ClusterLabs. Depending on the type that is selected, it is able to perform operation, such as start/stop for a given resource, monitor, validation, and so on. The Resource Agents that are supported are:

- LSB Resource Agents
- OCF Resource Agents
- The legacy Heartbeat Resource Agent

Cluster Glue is a set of libraries, utilities, and tools used in conjunction with Pacemaker/Heartbeat. It is the glue that basically puts everything together between the cluster resource manager (I am referring to Pacemaker) and the messaging layer (which could be Heartbeat). It is now managed as a separate component by the Linux-HA subproject, although it started as a component of Heartbeat. It has a number of interesting components:

- **Local Resource Manager (LRM)**: It acts as an interface between Pacemaker and the Resource Agent and is not cluster-aware. Its tasks include the processing of commands received from the CRM, passing them to the resource agent, and reporting these activities.
- **Shoot The Other Node In The Head (STONITH)**: It is a mechanism used for the purpose of node fencing by making a node that is considered dead by a cluster so that it can be removed from it and prevent any interaction risks.
- **hb_report**: It is an error reporting utility often used for bug fixing and isolation problems.
- **Cluster Plumbing Library**: It is a low-level intercluster communication library.



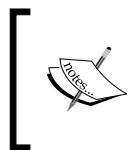
For more information related to Linux-HA the following link could be of help: <http://www.linux-ha.org/doc/users-guide/users-guide.html>



The next element is the Corosync cluster engine. It is a project derived from OpenAIS, which will be presented shortly. It is a Group Communication System with a set of features and implementations that try to offer high-availability support and is licensed under BSD. Its features include the following:

- An availability manager for the restarting of an application in case of failure.
- A quorum system that notifies about the state of a quorum and whether it's been achieved or not.
- A closed process group communication model with support for synchronization to replicate state machines.
- A configuration and statistics database that resides in the memory. It provides the ability to receive, retrieve, set, and change various notifications.

Next, we'll take a look at OpenAIS. It is the open implementation for **Application Interface Specification (AIS)** provided by **Service Availability Forum (SA or SA Forum** as it is also called). It represents an interface that provides high-availability support. The source code available in OpenAIS was refactored over time in OpenAIS and only remained SA Forum-specific APIs and in Corosync. It was also placed in all the core infrastructure components. OpenAIS is very similar to Heartbeat; it is, in fact, an alternative to it, which is industry standard-specific. It is also supported by Pacemaker.



More information about AIS can be found by referring to its Wikipedia page and the SA Forum web site at <http://www.saforum.org/page/16627~217404/Service-Availability-Forum-Application-Interface-Specification>.



Next is the `ocfs2-tools` package. It is a collection of utilities that enable the work to be done with the OCFS2 filesystem in the form of creating, debugging, repairing, or managing it. It includes tools that are very similar to the ones a Linux user is accustomed to, such as `mkfs.ocfs2`, `mount.ocfs2`, `fsck.ocfs2`, `tunefs.ocfs2`, and `debugfs.ocfs2`.

Oracle Cluster File System (OCFS) was the first shared disk filesystem developed by Oracle and was released under GNU General Public License. It was not a POSIX compliant filesystem, but this changed when OCFS2 appeared and was integrated into the Linux kernel. In time, it became a distributed lock manager capable of providing both high availability and high performance. It is now used in a variety of places, such as virtualization, database clusters, and middleware, and appliances. These are some of its most notable features:

- Optimized allocations
- REFLINKs
- Metadata checksums
- Indexed directories
- Extended attributes per inode
- User and group quotas
- Advanced security, such as SELinux and POSIX ACLs support
- Cluster-aware tools such as the ones mentioned previously and include `mkfs`, `tunefs`, `fsck`, `mount`, and `debugfs`
- In-built Clusterstack with a Distributed Lock Manager
- Journaling

- Variable block and cluster size
- Buffered, memory mapped, splice, direct, asynchronous I/Os
- Architecture and endian neutral

The `lksctp-tools` package is a Linux user space utility that includes a library and appropriate C language headers for the purpose of interaction with the SCTP interface. The Linux kernel has had support for SCTP since the 2.6 release, so the existence of the user space compatibility tools is no surprise for anyone. Lksctp offers access to the SCTP socket-based API. The implementation is made according to the IETF Internet draft available at <http://tools.ietf.org/html/draft-ietf-tsvwg-sctpsocket-15>. It provides a flexible and consistent method of developing socket-based applications that takes advantage of **Stream Control Transmission Protocol (SCTP)**.

SCTP is a message-oriented transport protocol. As a transport layer protocol, it runs over IPv4 or Ipv6 implementations and besides the functionality of TCP, it also provides support for these features:

- Multistreaming
- Message framing
- Multihoming
- Ordered and unordered message delivery
- Security and authentication

These special features are necessary for industry carrier graded systems and are used in fields such as telephony signaling.



More information about SCTP is available at <http://www.ietf.org/rfc/rfc2960.txt> and <http://www.ietf.org/rfc/rfc3286.txt>

Now, I will change the pace a bit and explain **monit**, a very small yet powerful utility to monitor and manage the system. It is very useful in automatic maintenance and repairing Unix systems, such as BSD distribution, various Linux distributions, and other platforms that can include OS X. It can be used for a large variety of tasks ranging from file monitoring, changes in filesystems, and interaction with event processes if various thresholds were passed.

It is easy to configure and control monit since all the configurations are based on a token-oriented syntax that is easy to grasp. Also, it offers a variety of logs and notifications about its activities. It also provides a web browser interface for easier access. So, having a general system resource manager, which is also easy to interact with, makes monit an option for a carrier graded Linux system. If you are interested in finding more about it, access the project's website at <http://monit.com/monit/>.

OpenIPMI is an implementation of **Intelligent Platform Management Interface (IPMI)** that tries to offer access to all the functionalities of IPMI and also offers abstractions for easier usage. It is comprised of two components:

- A kernel driver insertable in the Linux kernel
- A library that offers the abstraction functionality of IPMI and also provides access to various services used by an operating system

IPMI represents a set of computer interface specifications that try to reduce the total cost of ownership by offering an intelligent and autonomous system that is able to monitor and manage the capabilities of the host system. Here, we are referring to only about an operating system but also the firmware and CPU itself. The development of this intelligent interface was led by Intel and is now supported by an impressive number of companies.



More information about IPMI, OpenIPMI, and other supported IPMI drivers and functionality are available at <http://openipmi.sourceforge.net/> and <http://www.intel.com/content/www/us/en/servers/ipmi/ipmi-home.html>.

There are some packages that also should be present in the `meta-cgl` layer, but at the time of writing this chapter, they were still not available there. I will start with `mipv6-daemon-umip`, which tries to provide data distribution for **Mobile Internet Protocol version 6 (MIPv6)** daemons. UMIP is an open source Mobile IPv6 stack for Linux based on MIPL2 and maintains the latest kernel versions. The package is a set of patches for MIPL2 by the **UniverSAI playGround for Ipv6 (USAGI) Project**, which tries to offer industry ready quality for IPsec (for both IPv6 and IPv4 options) and IPv6 protocol stack implementations for the Linux system.



More information about UMIP is available at <http://umip.linux-ipv6.org/index.php?n>Main.Documentation>.

Makedumfile is a tool that offers the possibility of compressing the size of dump files and can also exclude memory pages that are not required for analysis. For some of the Linux distributions, it comes along with a package called `kexec-tools` that can be installed in your distribution using RPM, the package manager supported by the carrier graded specifications. It is quite similar to commands, such as `gzip` or `split`. The fact that it receives input only from files in ELF format, makes it the first choice for `kdumps`.

Another interesting project is `evlog`, a **Linux Event Logging system** for Enterprise-class systems. It also is compliant with POSIX standards and provides logging for a variety of forms that range from `printf` to `syslog` as well as other kernel and user space functions. The output events are available in a POSIX-compliant format. It also offers support while selecting logs that match certain defined filters or even register a special event format. These can only be notified about when the registered event filter is met. Its features certainly make this package interesting and are available at <http://evlog.sourceforge.net/>.

There are a number of other packages that could be included into the `meta-cgl` layer. Taking a look at the registered CGL distribution could help you understand the complexity of such a project. For easier access to this list, refer to <http://www.linuxfoundation.org/collaborate/workgroups/cgl/registered-distributions> in order to simplify the search procedure.

To interact with the `meta-cgl` layer, the first necessary step would be to make sure that all the interdependent layers are available. The latest information about how to build a carrier graded compatible Linux image is always available in the attached `README` file. I've also given you an example here for purpose of demonstrating it:

```
git clone git://git.yoctoproject.org/poky.git
cd ./poky
git clone git://git.yoctoproject.org/meta-openembedded.git
git clone git://git.enea.com/linux/meta-cgl.git
git clone git://git.yoctoproject.org/meta-qt3
git clone git://git.yoctoproject.org/meta-virtualization
git clone git://git.yoctoproject.org/meta-selinux
git clone git://git.yoctoproject.org/meta-cloud-services
git clone git://git.yoctoproject.org/meta-security
git clone https://github.com/joahf/meta-openclovis.git
```

Next, the build directory needs to be created and configured:

```
source oe-init-build-env -b ../build_cgl
```

Inside the `conf/bblayers.conf` file, these are the layers that need to be added:

```
meta-cgl/meta-cgl-common
meta-qt3
meta-openembedded/meta-networking
meta-openembedded/meta-filesystems
meta-openembedded/meta-oe
meta-openembedded/meta-perl
meta-virtualization
meta-openclovis
meta-selinux
meta-security
meta-cloud-services/meta-openstack
```

Inside the `conf/local.conf` file, the corresponding machine should be selected. I would suggest `qemuppc`, as well as the `DISTRO` variable that can be changed to `poky-cgl`. `BBMASK` should be made available due to duplication of recipes:

```
BBMASK = "meta-openembedded/meta-oe/recipes-support/multipath-tools"
```

Having all these place, the build process can start. The necessary command for this is:

```
bitbake core-image-cgl
```

Make sure that you have time to spend on this because the build could take a while, depending on the configuration of your host system.

Summary

In this chapter, you were presented with information about the specifications required for the Carrier Grade Linux and Linux Standard Base. Other options, such as Automotive Grade and Carrier Grade Virtualization, were also explained and in the end, support for the Yocto Project and a couple of demonstrations were shown to you to complete this learning process.

This is the last chapter of this book and I hope you've enjoyed the journey. Also, I hope I was able to pass on some of the information I have acquired on to you. Since we're at the end of this book, I must admit that I have also learned and gathered new information in the process of writing the book. I hope that you catch the Yocto bug as well and are also able to add your contributions to the Yocto Project and the open source community in general. I am confident that from now on, the embedded world holds fewer secrets for you. Make sure you shed some light about this topic on others too!

Module 2

Embedded Linux Projects Using Yocto Project Cookbook

*Over 70 hands-on recipes for professional embedded Linux developers
to optimize and boost their Yocto know-how*

1

The Build System

In this chapter, we will cover the following recipes:

- ▶ Setting up the host system
- ▶ Installing Poky
- ▶ Creating a build directory
- ▶ Building your first image
- ▶ Explaining the Freescale Yocto ecosystem
- ▶ Installing support for Freescale hardware
- ▶ Building Wandboard images
- ▶ Troubleshooting your Wandboard's first boot
- ▶ Configuring network booting for a development setup
- ▶ Sharing downloads
- ▶ Sharing the shared state cache
- ▶ Setting up a package feed
- ▶ Using build history
- ▶ Working with build statistics
- ▶ Debugging the build system

Introduction

The Yocto project (<http://www.yoctoproject.org/>) is an embedded Linux distribution builder that makes use of several other open source projects.

The Yocto project provides a reference build system for embedded Linux, called **Poky**, which has the **BitBake** and **OpenEmbedded-Core (OE-Core)** projects at its base. The purpose of Poky is to build the components needed for an embedded Linux product, namely:

- ▶ A bootloader image
- ▶ A Linux kernel image
- ▶ A root filesystem image
- ▶ Toolchains and **software development kits (SDKs)** for application development

With these, the Yocto project covers the needs of both system and application developers. When the Yocto project is used as an integration environment for bootloaders, the Linux kernel, and user space applications, we refer to it as system development.

For application development, the Yocto project builds SDKs that enable the development of applications independently of the Yocto build system.

The Yocto project makes a new release every six months. The latest release at the time of this writing is Yocto 1.7.1 Dizzy, and all the examples in this book refer to the 1.7.1 release.

A Yocto release comprises the following components:

- ▶ Poky, the reference build system
- ▶ A build appliance; that is, a VMware image of a host system ready to use Yocto
- ▶ An **Application Development Toolkit (ADT)** installer for your host system
- ▶ And for the different supported platforms:
 - Prebuilt toolchains
 - Prebuilt packaged binaries
 - Prebuilt images

The Yocto 1.7.1 release is available to download from <http://downloads.yoctoproject.org/releases/yocto/yocto-1.7.1/>.

Setting up the host system

This recipe will explain how to set up a host Linux system to use the Yocto project.

Getting ready

The recommended way to develop an embedded Linux system is using a native Linux workstation. Development work using virtual machines is discouraged, although they may be used for demo and test purposes.

Yocto builds all the components mentioned before from scratch, including the cross-compilation toolchain and the native tools it needs, so the Yocto build process is demanding in terms of processing power and both hard drive space and I/O.

Although Yocto will work fine on machines with lower specifications, for professional developer's workstations, it is recommended to use **symmetric multiprocessing (SMP)** systems with 8 GB or more system memory and a high capacity, fast hard drive. Build servers can employ distributed compilation, but this is out of the scope of this book. Due to different bottlenecks in the build process, there does not seem to be much improvement above 8 CPUs or around 16 GB RAM.

The first build will also download all the sources from the Internet, so a fast Internet connection is also recommended.

How to do it...

Yocto supports several distributions, and each Yocto release will document a list of the supported ones. Although the use of a supported Linux distribution is strongly advised, Yocto is able to run on any Linux system if it has the following dependencies:

- ▶ Git 1.7.8 or greater
- ▶ Tar 1.24 or greater
- ▶ Python 2.7.3 or greater (but not Python 3)

Yocto also provides a way to install the correct version of these tools by either downloading a *buildtools-tarball* or building one on a supported machine. This allows virtually any Linux distribution to be able to run Yocto, and also makes sure that it will be possible to replicate your Yocto build system in the future. This is important for embedded products with long-term availability requirements.

This book will use the Ubuntu 14.04 **Long-Term Stable (LTS)** Linux distribution for all examples. Instructions to install on other Linux distributions can be found on the *Supported Linux Distributions* section of the *Yocto Project Development Manual*, but the examples will only be tested with Ubuntu 14.04 LTS.

To make sure you have the required package dependencies installed for Yocto and to follow the examples in the book, run the following command from your shell:

```
$ sudo apt-get install gawk wget git-core diffstat unzip texinfo gcc-multilib build-essential chrpath socat libsdl1.2-dev xterm make xsltproc docbook-utils fop dblatex xmlto autoconf automake libtool libglib2.0-dev python-gtk2 bsdmainutils screen
```

How it works...

The preceding command will use `apt-get`, the **Advanced Packaging Tool (APT)**, command-line tool. It is a frontend of the `dpkg` package manager that is included in the Ubuntu distribution. It will install all the required packages and their dependencies to support all the features of the Yocto project.

There's more...

If build times are an important factor for you, there are certain steps you can take when preparing your disks to optimize them even further:

- ▶ Place the build directories on their own disk partition or a fast external drive.
- ▶ Use the ext4 filesystem but configure it not to use journaling on your Yocto-dedicated partitions. Be aware that power losses may corrupt your build data.
- ▶ Mount the filesystem in such a way that read times are not written/recorded on file reads, disable write barriers, and delay committing filesystem changes with the following mount options:
`noatime,barrier=0,commit=6000.`
- ▶ Do not build on network-mounted drives.

These changes reduce the data integrity safeguards, but with the separation of the build directories to their own disk, failures would only affect temporary build data, which can be erased and regenerated.

See also

- ▶ The complete Yocto project installation instructions for Ubuntu and other supported distributions can be found on the *Yocto Project Reference Manual* at <http://www.yoctoproject.org/docs/1.7.1/ref-manual/ref-manual.html>

Installing Poky

This recipe will explain how to set up your host Linux system with Poky, the Yocto project reference system.

Getting ready

Poky uses the OpenEmbedded build system, and as such, uses the BitBake tool, a task scheduler written in Python which forked from Gentoo's Portage tool. You can think of BitBake as the make utility in Yocto. It will parse the configuration and recipe metadata, schedule a task list, and run through it.

BitBake is also the command-line interface to Yocto.

Poky and BitBake are two of the open source projects used by Yocto. The Poky project is maintained by the Yocto community. You can download Poky from its Git repository at <http://git.yoctoproject.org/cgit.cgi/poky/>.

Development discussions can be followed and contributed to by visiting the development mailing list at <https://lists.yoctoproject.org/listinfo/poky>.

BitBake, on the other hand, is maintained by both the Yocto and OpenEmbedded communities, as the tool is used by both. BitBake can be downloaded from its Git repository at <http://git.openembedded.org/bitbake/>.

Development discussions can be followed and contributed to by visiting the development mailing list at <http://lists.openembedded.org/mailman/listinfo/bitbake-devel>.

The Poky build system only supports virtualized QEMU machines for the following architectures:

- ▶ ARM (qemuarm)
- ▶ x86 (qemux86)
- ▶ x86-64 (qemux86-64)
- ▶ PowerPC (qemuppc)
- ▶ MIPS (qemumips, qemumips64)

Apart from these, it also supports some reference hardware **Board Support Packages (BSPs)**, representative of the architectures just listed. These are those BSPs:

- ▶ Texas Instruments Beaglebone (beaglebone)
- ▶ Freescale MPC8315E-RDB (mpc8315e-rdb)
- ▶ Intel x86 based PCs and devices (genericx86 and genericx86-64)
- ▶ Ubiquiti Networks EdgeRouter Lite (edgerouter)

To develop on different hardware, you will need to complement Poky with hardware-specific Yocto layers. This will be covered later on.

How to do it...

The Poky project incorporates a stable BitBake release, so to get started with Yocto, we only need to install Poky in our Linux host system.



Note that you can also install BitBake independently through your distribution's package management system. This is not recommended and can be a source of problems, as BitBake needs to be compatible with the metadata used in Yocto. If you have installed BitBake from your distribution, please remove it.

The current Yocto release is 1.7.1, or Dizzy, so we will install that into our host system. We will use the `/opt/yocto` folder as the installation path:

```
$ sudo install -o $(id -u) -g $(id -g) -d /opt/yocto  
$ cd /opt/yocto  
$ git clone --branch dizzy git://git.yoctoproject.org/poky
```

How it works...

The previous instructions will use Git (the source code management system command-line tool) to clone the Poky repository, which includes BitBake, into a new `poky` directory on our current path, and point it to the Dizzy stable branch.

There's more...

Poky contains three metadata directories, `meta`, `meta-yocto`, and `meta-yocto-bsp`, as well as a template metadata layer, `meta-skeleton`, that can be used as a base for new layers. Poky's three metadata directories are explained here:

- ▶ `meta`: This directory contains the OpenEmbedded-Core metadata, which supports the ARM, x86, x86-64, PowerPC, MIPS, and MIPS64 architectures and the QEMU emulated hardware. You can download it from its Git repository at <http://git.openembedded.org/openembedded-core/>.
Development discussions can be followed and contributed to by visiting the development mailing list at <http://lists.openembedded.org/mailman/listinfo/openembedded-core>.
- ▶ `meta-yocto`: This contains Poky's distribution-specific metadata.
- ▶ `meta-yocto-bsp`: This contains metadata for the reference hardware boards.

See also

- ▶ There is documentation about Git, the distributed version control system, at <http://git-scm.com/doc>

Creating a build directory

Before building your first Yocto image, we need to create a `build` directory for it.

The build process, on a host system as outlined before, can take up to one hour and need around 20 GB of hard drive space for a console-only image. A graphical image, like `core-image-sato`, can take up to 4 hours for the build process and occupy around 50 GB of space.

How to do it...

The first thing we need to do is create a `build` directory for our project, where the build output will be generated. Sometimes, the `build` directory may be referred to as the project directory, but `build` directory is the appropriate Yocto term.

There is no right way to structure the `build` directories when you have multiple projects, but a good practice is to have one `build` directory per architecture or machine type. They can all share a common `downloads` folder, and even a shared state cache (this will be covered later on), so keeping them separate won't affect the build performance, but it will allow you to develop on multiple projects simultaneously.

To create a `build` directory, we use the `oe-init-build-env` script provided by Poky. The script needs to be sourced into your current shell, and it will set up your environment to use the OpenEmbedded/Yocto build system, including adding the BitBake utility to your path. You can specify a `build` directory to use or it will use `build` by default. We will use `qemuarm` for this example.

```
$ cd /opt/yocto/poky  
$ source oe-init-build-env qemuarm
```

The script will change to the specified directory.



As `oe-init-build-env` only configures the current shell, you will need to source it on every new shell. But, if you point the script to an existing `build` directory, it will set up your environment but won't change any of your existing configurations.



BitBake is designed with a client/server abstraction, so we can also start a memory resident server and connect a client to it. With this setup, loading cache and configuration information each time is avoided, which saves some overhead. To run a memory resident BitBake that will always be available, you can use the `oe-init-build-env-memres` script as follows:

```
$ source oe-init-build-env-memres 12345 qemuarm
```

Here 12345 is the local port to be used.

Do not use both BitBake flavors simultaneously, as this can be a source of problems.

You can then kill the memory resident BitBake by executing the following command:

```
$ bitbake -m
```

How it works...

Both scripts call the `scripts/oe-setup-buildDir` script inside the `poky` directory to create the `build` directory.

On creation, the `build` directory contains a `conf` directory with the following three files:

- ▶ `bblayers.conf`: This file lists the metadata layers to be considered for this project.
- ▶ `local.conf`: This file contains the project-specific configuration variables. You can set common configuration variables to different projects with a `site.conf` file, but this is not created by default.

- ▶ `templateconf.cfg`: This file contains the directory that includes the template configuration files used to create the project. By default it uses the one pointed to by the `templateconf` file in your Poky installation directory, which is `meta-yocto/conf` by default.

To start a build from scratch, that's all the build directory needs.

Erasing everything apart from these files will recreate your build from scratch.

```
$ cd /opt/yocto/poky/qemuarm  
$ rm -Rf tmp sstate-cache
```

There's more...

You can specify a different template configuration file to use when you create your build directory using the `TEMPLATECONF` variable; for example:

```
$ TEMPLATECONF=meta-custom/config source oe-init-build-env <build-dir>
```

The `TEMPLATECONF` variable needs to refer to a directory containing templates for both `local.conf` and `bblayer.conf`, but named `local.conf.sample` and `bblayers.conf.sample`.

For our purposes, we can use the unmodified default project configuration files.

Building your first image

Before building our first image, we need to decide what type of image we want to build. This recipe will introduce some of the available Yocto images and provide instructions to build a simple image.

Getting ready

Poky contains a set of default target images. You can list them by executing the following commands:

```
$ cd /opt/yocto/poky  
$ ls meta*/recipes*/images/*.bb
```

A full description of the different images can be found on the *Yocto Project Reference Manual*. Typically, these default images are used as a base and customized for your own project needs. The most frequently used base default images are:

- ▶ `core-image-minimal`: This is the smallest BusyBox-, sysvinit-, and udev-based console-only image
- ▶ `core-image-full-cmdline`: This is the BusyBox-based console-only image with full hardware support and a more complete Linux system, including bash
- ▶ `core-image-lsb`: This is a console-only image that is based on Linux Standard Base compliance
- ▶ `core-image-x11`: This is the basic X11 Windows-system-based image with a graphical terminal
- ▶ `core-image-sato`: This is the X11 Window-system-based image with a SATO theme and a GNOME Mobile desktop environment
- ▶ `core-image-weston`: This is a Wayland protocol and Weston reference compositor-based image

You will also find images with the following suffixes:

- ▶ `dev`: These images are suitable for development work, as they contain headers and libraries.
- ▶ `sdk`: These images include a complete SDK that can be used for development on the target.
- ▶ `initramfs`: This is an image that can be used for a RAM-based root filesystem, which can optionally be embedded with the Linux kernel.

How to do it...

To build an image, we need to configure the `MACHINE` we are building it for and pass its name to BitBake. For example, for the `qemuarm` machine, we would run the following:

```
$ cd /opt/yocto/poky/qemuarm  
$ MACHINE=qemuarm bitbake core-image-minimal
```

Or we could export the `MACHINE` variable to the current shell environment with the following:

```
$ export MACHINE=qemuarm
```

But the preferred and persistent way to do it is to edit the `conf/local.conf` configuration file to change the default machine to `qemuarm`:

```
- #MACHINE ?= "qemuarm"  
+ MACHINE ?= "qemuarm"
```

Then you can just execute the following:

```
$ bitbake core-image-minimal
```

How it works...

When you pass a target recipe to BitBake, it first parses the following configuration files:

- ▶ conf/bblayers.conf: This file is used to find all the configured layers
- ▶ conf/layer.conf: This file is used on each configured layer
- ▶ meta/conf/bitbake.conf: This file is used for its own configuration
- ▶ conf/local.conf: This file is used for any other configuration the user may have for the current build
- ▶ conf/machine/<machine>.conf: This file is the machine configuration; in our case, this is qemuart.conf
- ▶ conf/distro/<distro>.conf: This file is the distribution policy; by default, this is the poky.conf file

And then BitBake parses the target recipe that has been provided and its dependencies. The outcome is a set of interdependent tasks that BitBake will then execute in order.

There's more...

Most developers won't be interested in keeping the whole build output for every package, so it is recommended to configure your project to remove it with the following configuration in your conf/local.conf file:

```
INHERIT += "rm_work"
```

But at the same time, configuring it for all packages means that you won't be able to develop or debug them.

You can add a list of packages to exclude from cleaning by adding them to the RM_WORK_EXCLUDE variable. For example, if you are going to do BSP work, a good setting might be:

```
RM_WORK_EXCLUDE += "linux-yocto u-boot"
```

Remember that you can use a custom template local.conf.sample configuration file in your own layer to keep these configurations and apply them for all projects so that they can be shared across all developers.

Once the build finishes, you can find the output images on the tmp/deploy/images/qemuart directory inside your build directory.

By default, images are not erased from the `deploy` directory, but you can configure your project to remove the previously built version of the same image by adding the following to your `conf/local.conf` file:

```
RM_OLD_IMAGE = "1"
```

You can test run your images on the QEMU emulator by executing this:

```
$ runqemu qemuarm core-image-minimal
```

The `runqemu` script included in Poky's `scripts` directory is a launch wrapper around the QEMU machine emulator to simplify its usage.

Explaining the Freescale Yocto ecosystem

As we saw, Poky metadata starts with the `meta`, `meta-yocto`, and `meta-yocto-bsp` layers, and it can be expanded by using more layers.

An index of the available OpenEmbedded layers that are compatible with the Yocto project is maintained at <http://layers.openembedded.org/>.

An embedded product's development usually starts with hardware evaluation using a manufacturer's reference board design. Unless you are working with one of the reference boards already supported by Poky, you will need to extend Poky to support your hardware.

Getting ready

The first thing to do is to select which base hardware your design is going to be based on. We will use a board that is based on a Freescale i.MX6 **System on Chip (SoC)** as a starting point for our embedded product design.

This recipe gives an overview of the support for Freescale hardware in the Yocto project.

How to do it...

The SoC manufacturer (in this case, Freescale) has a range of reference design boards for purchase, as well as official Yocto-based software releases. Similarly, other manufacturers that use Freescale's SoCs offer reference design boards and their own Yocto-based software releases.

Selecting the appropriate hardware to base your design on is one of the most important design decisions for an embedded product. Depending on your product needs, you will decide to either:

- ▶ Use a production-ready board, like a **single-board computer (SBC)**
- ▶ Use a module and build your custom carrier board around it
- ▶ Use Freescale's SoC directly and design your own board

Most of the times, a production-ready board will not match the specific requirements of an professional embedded system, and the process of designing a complete carrier board using Freescale's SoC would be too time consuming. So, using an appropriate module that already solves the most technically challenging design aspects is a common choice.

Some of the characteristics that are important to consider are:

- ▶ Industrial temperature ranges
- ▶ Power management
- ▶ Long-term availability
- ▶ Precertified wireless and Bluetooth (if applicable)

The Yocto community layers that support Freescale-based boards are called `meta-fsl-arm` and `meta-fsl-arm-extras`. The selection of boards that are supported on `meta-fsl-arm` is limited to Freescale reference designs, which would be the starting point if you are considering designing your own carrier board around Freescale's SoC. Boards from other vendors are maintained on the `meta-fsl-arm-extras` layer.

There are other embedded manufacturers that use `meta-fsl-arm`, but they have not integrated their boards in the `meta-fsl-arm-extras` community layer. These manufacturers will keep their own BSP layers, which depend on `meta-fsl-arm`, with specific support for their hardware. An example of this is Digi International and its ConnectCore 6 module, which is based on the i.MX6 SoC.

How it works...

To understand Freescale Yocto ecosystem, we need to start with the Freescale community BSP, comprising the `meta-fsl-arm` layer with support for Freescale reference boards, and its companion, `meta-fsl-arm-extra`, with support for boards from other vendors, and its differences with the official Freescale Yocto releases that Freescale offers for their reference designs.

There are some key differences between the community and Freescale Yocto releases:

- ▶ Freescale releases are developed internally by Freescale without community involvement and are used for BSP validation on Freescale reference boards.
- ▶ Freescale releases go through an internal QA and validation test process, and they are maintained by Freescale support.
- ▶ Freescale releases for a specific platform reach a maturity point, after which they are no longer worked on. At this point, all the development work has been integrated into the community layer and the platforms are further maintained by the Freescale BSP community.
- ▶ Freescale Yocto releases are not Yocto compatible, while the community release is.

Freescale's engineering works very closely with the Freescale BSP community to make sure that all development in their official releases is integrated in the community layer in a reliable and quick manner.

Usually, the best option is to use the Freescale BSP community release but stay with the U-Boot and Linux kernel versions that were released as part of the manufacturer's stable BSP release.

This effectively means that you get the latest updates to the Linux kernel and U-Boot from the manufacturer while simultaneously getting the latest updates to the root filesystem from the community, extending the lifetime of your product, and making sure you are up to date with applications, bug fixes, and security updates.

This takes advantage of the manufacturer's QA process for the system components that are closer to the hardware, and makes it possible to use the manufacturer's support while simultaneously getting user space updates from the community. The Freescale BSP community is also very responsive and active, so problems can usually be worked on with them to benefit all parts.

There's more...

The Freescale BSP community extends Poky with the following layers:

- ▶ `meta-fsl-arm`: This is the community layer that supports Freescale reference designs. It has a dependency on OpenEmbedded-Core. Machines in this layer will be maintained even after Freescale stops active development on them. You can download `meta-fsl-arm` from its Git repository at <http://git.yoctoproject.org/cgit/cgit.cgi/meta-fsl-arm/>.

Development discussions can be followed and contributed to by visiting the development mailing list at <https://lists.yoctoproject.org/listinfo/meta-freescale>.

The `meta-fsl-arm` layer pulls both the Linux kernel and the U-Boot source from Freescale's repositories using the following links:

- **Freescale Linux kernel Git repository:** <http://git.freescale.com/git/cgit.cgi/imx/linux-2.6-imx.git/>
- **Freescale U-Boot Git repository:** <http://git.freescale.com/git/cgit.cgi/imx/uboot-imx.git/>

Other Linux kernel and U-Boot versions are available, but keeping the manufacturer's supported version is recommended.

The `meta-fsl-arm` layer includes Freescale's proprietary binaries to enable some hardware features – most notably its hardware graphics, multimedia, and encryption capabilities. To make use of these capabilities, the end user needs to accept Freescale's **End-User License Agreement (EULA)**, which is included in the `meta-fsl-arm` layer. To accept the license, the following line needs to be added to the project's `conf/local.conf` configuration file:

```
ACCEPT_FSL_EULA = "1"
```

- ▶ `meta-fsl-arm-extra`: This layer adds support for other community-maintained boards; for example, the Wandboard. To download the layer's content, you may visit <https://github.com/Freescale/meta-fsl-arm-extra/>.
- ▶ `meta-fsl-demos`: This layer adds a metadata layer for demonstration target images. To download the layer's content, you may visit <https://github.com/Freescale/meta-fsl-demos>.

Freescale uses another layer on top of the layers above for their official software releases: `meta-fsl-bsp-release`.

- ▶ `meta-fsl-bsp-release`: This is a Freescale-maintained layer that is used in the official Freescale software releases. It contains modifications to both `meta-fsl-arm` and `meta-fsl-demos`. It is not part of the community release.

See also

- ▶ For more information, refer to the FSL community BSP release notes available at <http://freescale.github.io/doc/release-notes/1.7/>

Installing support for Freescale hardware

In this recipe, we will install the community Freescale BSP Yocto release that adds support for Freescale hardware to our Yocto installation.

Getting ready

With so many layers, manually cloning each of them and adding them to your project's `conf/bblayers.conf` file is cumbersome. The community is using the `repo` tool developed by Google for their community Android to ease the installation of Yocto.

To install `repo` in your host system, type in the following commands:

```
$ sudo curl http://commondatastorage.googleapis.com/git-repo-downloads/repo > /usr/local/sbin/repo  
$ sudo chmod a+x /usr/local/sbin/repo
```

The `repo` tool is a Python utility that parses an XML file, called `manifest`, with a list of Git repositories. The `repo` tool is then used to manage those repositories as a whole.

How to do it...

For example, we will use `repo` to download all the repositories listed in the previous recipe to our host system. For that, we will point it to the Freescale community BSP manifest for the Dizzy release:

```
<?xml version="1.0" encoding="UTF-8"?>
<manifest>
    <default sync-j="4" revision="master"/>
    <remote fetch="git://git.yoctoproject.org" name="yocto"/>
    <remote fetch="git://github.com/Freescale" name="freescale"/>
    <remote fetch="git://git.openembedded.org" name="oe"/>
    <project remote="yocto" revision="dizzy" name="poky"
path="sources/poky"/>
    <project remote="yocto" revision="dizzy" name="meta-fsl-arm"
path="sources/meta-fsl-arm"/>
    <project remote="oe" revision="dizzy" name="meta-openembedded"
path="sources/meta-openembedded"/>
    <project remote="freescale" revision="dizzy" name="fsl-
community-bsp-base" path="sources/base">
        <copyfile dest="README" src="README"/>
        <copyfile dest="setup-environment" src="setup-
environment"/>
    </project>
    <project remote="freescale" revision="dizzy" name="meta-fsl-arm-
extra" path="sources/meta-fsl-arm-extra"/>
    <project remote="freescale" revision="dizzy" name="meta-fsl-
demos" path="sources/meta-fsl-demos"/>
    <project remote="freescale" revision="dizzy"
name="Documentation" path="sources/Documentation"/>
</manifest>>
```

The manifest file shows all the installation paths and repository sources for the different components that are going to be installed.

How it works...

The manifest file is a list of the different layers that are needed for the Freescale community BSP release. We can now use `repo` to install it. Run the following:

```
$ mkdir /opt/yocto/fsl-community-bsp
$ cd /opt/yocto/fsl-community-bsp
$ repo init -u https://github.com/Freescale/fsl-community-bsp-
platform -b dizzy
$ repo sync
```

You can optionally pass a `-jN` argument to sync if you have a multicore machine for multithreaded operations; for example, you could pass `repo sync -j8` in an 8-core host system.

There's more...

To list the hardware boards supported by the different layers, we may run:

```
$ ls sources/meta-fsl*/conf/machine/*.conf
```

And to list the newly introduced target images, use the following:

```
$ ls sources/meta-fsl*/recipes*/images/*.bb
```

The community Freescale BSP release introduces the following new target images:

- ▶ `fsl-image-mfgtool-initramfs`: This is a small, RAM-based `initramfs` image used with the Freescale manufacturing tool
- ▶ `fsl-image-multimedia`: This is a console-only image that includes the `gstreamer` multimedia framework over the framebuffer, if applicable
- ▶ `fsl-image-multimedia-full`: This is an extension of `fsl-image-multimedia`, but extends the `gstreamer` multimedia framework to include all available plugins
- ▶ `fsl-image-machine-test`: This is an extension on `fsl-image-multimedia-full` for testing and benchmarking
- ▶ `qte-in-use-image`: This is a graphical image that includes support for Qt4 over the framebuffer
- ▶ `qt-in-use-image`: This is a graphical image that includes support for Qt4 over the X11 Windows system

See also

- ▶ Instructions to use the `repo` tool, including using `repo` with proxy servers, can be found in the Android documentation at <https://source.android.com/source/downloading.html>

Building Wandboard images

Building images for one of the supported boards (for example, `Wandboard Quad`) follows the same process we described earlier for the QEMU machines, with the exception of using the `setup-environment` script, which is a wrapper around `oe-init-build-env`.

How to do it...

To build an image for the wandboard-quad machine, use the following commands:

```
$ cd /opt/yocto/fsl-community-bsp  
$ mkdir -p wandboard-quad  
$ MACHINE=wandboard-quad source setup-environment wandboard-quad  
$ bitbake core-image-minimal
```



The current version of the setup-environment script only works if the build directory is under the installation folder; in our case, /opt/yocto/fsl-community-bsp.

How it works...

The setup-environment script will create a build directory, set up the MACHINE variable, and prompt you to accept the Freescale EULA as described earlier. Your conf/local.conf configuration file will be updated both with the specified machine and the EULA acceptance variable.



Remember that if you close your terminal session, you will need to set up the environment again before being able to use BitBake. You can safely rerun the setup-environment script as seen previously, as it will not touch an existing conf/local.conf file. Run the following:

```
$ cd /opt/yocto/fsl-community-bsp/  
$ source setup-environment wandboard-quad
```

The resulting image, core-image-minimal.sdcard, which is created inside the build directory, can be programmed into a microSD card, inserted into the primary slot in the Wandboard CPU board, and booted using the following commands:

```
$ cd /opt/yocto/fsl-community-bsp/wandboard-quad/tmp/deploy/images/wandboard-quad/  
$ sudo dd if=core-image-minimal.sdcard of=/dev/sdN bs=1M && sync
```

Here, /dev/sdN corresponds to the device node assigned to the microSD card in your host system.



Be careful when running the dd command, as it could harm your machine. You need to be absolutely sure that the sdN device corresponds to your microSD card and not a drive on your development machine.

See also

- ▶ You can find more information regarding the `repo` tool on Android's documentation at <https://source.android.com/source/using-repo.html>

Troubleshooting your Wandboard's first boot

If you have problems booting your image, follow this recipe to troubleshoot.

Getting ready

1. Without the microSD card inserted, plug in a microUSB-to-USB cable to the USB OTG interface of your Wandboard. Check the `lsusb` utility on your Linux host to see whether the Wandboard appears as follows:

```
Bus 002 Device 006: ID 15a2:0054 Freescale Semiconductor, Inc.  
    i.MX6Q SystemOnChip in RecoveryMode
```

If you don't see this, try a different power supply. It should be 5V, 10W.

2. Make sure you connect a NULL modem serial cable between the RS232 connector in your Wandboard target and a serial port on your Linux host. Then open a terminal program like minicom with the following:

```
$ minicom -D /dev/ttys0 -b 115200
```

 You will need to add your user to the dialout group, or try to run the command as sudo. This should open a 115200 8N1 serial connection. The serial device may vary in your Linux host. For example, a USB-to-serial adapter may be detected as `/dev/ttys0`. Also, make sure both hardware and software flow control are disabled.

How to do it...

1. Insert the microSD card image to the module slot, not the base board, as the latter is only used for storage and not for booting, and power it. You should see the U-Boot banner in the minicom session output.
2. If not, you may have a problem with the serial communication. By default, the Ethernet interface in the FSL community BSP image is configured to request an address by DHCP, so you can use that to connect to the target.

Make sure you have a DHCP server running on the test network where the target is.

You can use a packet sniffer like **Wireshark** to capture a network trace on your Linux host and filter packages like the `bootp` protocol. At least, you should see some broadcasts from your target, and if you use an Ethernet hub, you should also see the DHCP replies.

Optionally, you can log in to your DHCP server and check the logs to see if a new IP address has been assigned. If you see an IP address being assigned, you might want to consider adding an SSH server, like **Dropbear**, to your core-image-minimal image so that you can establish a network connection with the target. You can do this by adding the following line to the `conf/local.conf` configuration file:

```
IMAGE_INSTALL_append = " dropbear"
```

Note the space after the initial quote.

After building and reprogramming, you can then start an SSH session to the Wandboard from your Linux host with:

```
$ ssh root@<ip_address>
```

The connection should automatically log in without a password prompt.

3. Try to program the default microSD card images from <http://www.wandboard.org/index.php/downloads> to make sure the hardware and your setup is valid.
4. Try to reprogram your microSD card. Make sure you are using the correct images for your board (for example, do not mix dual and quad images). Also, try different cards and card readers.

These steps will have your Wandboard start booting, and you should have some output in your serial connection.

There's more...

If everything else fails, you can verify the position of the bootloader on your microSD card. You can dump the contents of the first blocks of your microSD card with:

```
$ sudo dd if=/dev/sdN of=/tmp/sdcard.img count=10
```

You should see a U-Boot header at offset 0x400. That's the offset where the i.MX6 boot ROM will be looking for the bootloader when bootstrapped to boot from the microSD interface. Use the following commands:

```
$ head /tmp/sdcard.img | hexdump  
0000400 00d1 4020 0000 1780 0000 0000 f42c 177f
```

You can recognize the U-Boot header by dumping the U-Boot image from your build. Run the following commands:

```
$ head u-boot-wandboard-quad.imx | hexdump  
0000000 00d1 4020 0000 1780 0000 0000 f42c 177f
```

Configuring network booting for a development setup

Most professional i.MX6 boards will have an internal **embedded MMC (eMMC)** flash memory, and that would be the recommended way to boot firmware. The Wandboard is not really a product meant for professional use, so it does not have one. But neither the eMMC nor the microSD card are ideal for development work, as any system change would involve a reprogramming of the firmware image.

Getting ready

The ideal setup for development work is to use both TFTP and NFS servers in your host system and to only store the U-Boot bootloader in either the eMMC or a microSD card. With this setup, the bootloader will fetch the Linux kernel from the TFTP server and the kernel will mount the root filesystem from the NFS server. Changes to either the kernel or the root filesystem are available without the need to reprogram. Only bootloader development work would need you to reprogram the physical media.

Installing a TFTP server

If you are not already running a TFTP server, follow the next steps to install and configure a TFTP server on your Ubuntu 14.04 host:

```
$ sudo apt-get install tftpd-hpa
```

The `tftpd-hpa` configuration file is installed in `/etc/default/tftpd-hpa`. By default, it uses `/var/lib/tftpboot` as the root TFTP folder. Change the folder permissions to make it accessible to all users using the following command:

```
$ sudo chmod 1777 /var/lib/tftpboot
```

Now copy the Linux kernel and device tree from your build directory as follows:

```
$ cd /opt/yocto/fsl-community-bsp/wandboard-
  quad/tmp/deploy/images/wandboard-quad/
$ cp zImage-wandboard-quad.bin zImage-imx6q-wandboard.dtb
  /var/lib/tftpboot
```

Installing an NFS server

If you are not already running an NFS server, follow the next steps to install and configure one on your Ubuntu 14.04 host:

```
$ sudo apt-get install nfs-kernel-server
```

We will use the `/nfsroot` directory as the root for the NFS server, so we will "untar" the target's root filesystem from our Yocto build directory in there:

```
$ sudo mkdir /nfsroot  
$ cd /nfsroot  
$ sudo tar xvf /opt/yocto/fsl-community-bsp/wandboard-  
quad/tmp/deploy/images/wandboard-quad/core-image-minimal-wandboard-  
quad.tar.bz2
```

Next, we will configure the NFS server to export the `/nfsroot` folder:

```
/etc(exports:  
/nfsroot/ *(rw,no_root_squash,async,no_subtree_check)
```

We will then restart the NFS server for the configuration changes to take effect:

```
$ sudo service nfs-kernel-server restart
```

How to do it...

Boot the Wandboard and stop at the U-Boot prompt by pressing any key on the serial console. Then run through the following steps:

1. Get an IP address by DHCP:

```
> dhcp
```

Alternatively, you can configure a static IP address with:

```
> setenv ipaddr <static_ip>
```

2. Configure the IP address of your host system, where the TFTP and NFS servers have been set up:

```
> setenv serverip <host_ip>
```

3. Configure the root filesystem mount:

```
> setenv nfsroot /nfsroot
```

4. Configure the Linux kernel and device tree filenames:

```
> setenv image zImage-wandboard-quad.bin
```

```
> setenv fdt_file zImage-imx6q-wandboard.dtb
```

5. If you have configured a static IP address, you need to disable DHCP on boot by running:

```
> setenv ip_dyn no
```

6. Save the U-Boot environment to the microSD card:

```
> saveenv
```

7. Perform a network boot:

```
> run netboot
```

The Linux kernel and device tree will be fetched from the TFTP server, and the root filesystem will be mounted by the kernel from the NFS share after getting a DHCP address from your network (unless using static IP addresses).

You should be able to log in with the root user without a password prompt.

Sharing downloads

You will usually work on several projects simultaneously, probably for different hardware platforms or different target images. In such cases, it is important to optimize the build times by sharing downloads.

Getting ready

The build system runs a search for downloaded sources in a number of places:

- ▶ It tries the local `downloads` folder.
- ▶ It looks into the configured premirrors, which are usually local to your organization.
- ▶ It then tries to fetch from the upstream source as configured in the package recipe.
- ▶ Finally, it checks the configured mirrors. Mirrors are public alternate locations for the source.

If a package source is not found in any of the these four, the package build will fail with an error. Build warnings are also issued when upstream fetching fails and mirrors are tried, so that the upstream problem can be looked at.

The Yocto project maintains a set of mirrors to isolate the build system from problems with the upstream servers. However, when adding external layers, you could be adding support for packages that are not in the Yocto project's mirror servers, or other configured mirrors, so it is recommended that you keep a local premirror to avoid problems with source availability.

The default Poky setting for a new project is to store the downloaded package sources on the current `build` directory. This is the first place the build system will run a search for source downloads. This setting can be configured in your project's `conf/local.conf` file with the `DL_DIR` configuration variable.

How to do it...

To optimize the build time, it is recommended to keep a shared downloads directory between all your projects. The `setup-environment` script of the `meta-fsl-arm` layer changes the default `DL_DIR` to the `fsl-community-bsp` directory created by the `repo` tool. With this setup, the `downloads` folder will already be shared between all the projects in your host system. It is configured as:

```
DL_DIR ?= "${BSPDIR}/downloads/"
```

A more scalable setup (for instance, for teams that are remotely distributed) is to configure a premirror. For example, adding the following to your `conf/local.conf` file:

```
INHERIT += "own-mirrors"  
SOURCE_MIRROR_URL = "http://example.com/my-source-mirror"
```

A usual setup is to have a build server serve its `downloads` directory. The build server can be configured to prepare tarballs of the Git directories to avoid having to perform Git operations from upstream servers. This setting in your `conf/local.conf` file will affect the build performance, but this is usually acceptable in a build server. Add the following:

```
BB_GENERATE_MIRROR_TARBALLS = "1"
```

An advantage of this setup is that the build server's `downloads` folder can also be backed up to guarantee source availability for your products in the future. This is especially important in embedded products with long-term availability requirements.

In order to test this setup, you may check to see whether a build is possible just by using the premirrors with the following:

```
BB_FETCH_PREMIRRORONLY = "1"
```

This setting in your `conf/local.conf` file can also be distributed across the team with the `TEMPLATECONF` variable during the project's creation.

Sharing the shared state cache

The Yocto project builds everything from source. When you create a new project, only the configuration files are created. The build process then compiles everything from scratch, including the cross-compilation toolchain and some native tools important for the build.

This process can take a long time, and the Yocto project implements a shared state cache mechanism that is used for incremental builds with the aim to build only the strictly necessary components for a given change.

For this to work, the build system calculates a checksum of the given input data to a task. If the input data changes, the task needs to be rebuilt. In simplistic terms, the build process generates a run script for each task that can be checksummed and compared. It also keeps track of a task's output, so that it can be reused.

A package recipe can modify the shared state caching to a task; for example, to always force a rebuild by marking it as `nostamp`. A more in-depth explanation of the shared state cache mechanism can be found in the *Yocto Project Reference Manual* at <http://www.yoctoproject.org/docs/1.7.1/ref-manual/ref-manual.html>.

How to do it...

By default, the build system will use a shared state cache directory called `sstate-cache` on your build directory to store the cached data. This can be changed with the `SSTATE_DIR` configuration variable in your `conf/local.conf` file. The cached data is stored in directories named with the first two characters of the hash. Inside, the filenames contain the whole task checksum, so the cache validity can be ascertained just by looking at the filename. The build process set scene tasks will evaluate the cached data and use it to accelerate the build if valid.

When you want to start a build from a clean state, you need to remove both the `sstate-cache` directory and the `tmp` directory.

You can also instruct BitBake to ignore the shared state cache by using the `--no-setscene` argument when running it.

It's a good practice to keep backups of clean shared state caches (for example, from a build server), which can be used in case of shared state cache corruption.

There's more...

Sharing a shared state cache is possible; however, it needs to be approached with care. Not all changes are detected by the shared state cache implementation, and when this happens, some or all of the cache needs to be invalidated. This can cause problems when the state cache is being shared.

The recommendation in this case depends on the use case. Developers working on Yocto metadata should keep the shared state cache as default, separated per project.

However, validation and testing engineers, kernel and bootloader developers, and application developers would probably benefit from a well-maintained shared state cache.

To configure an NFS share drive to be shared among the development team to speed up the builds, you can add the following to your `conf/local.conf` configuration file:

```
SSTATE_MIRRORS ?= "\\\nfile://.* file:///nfs/local/mount/sstate/PATH"
```

The expression `PATH` in this example will get substituted by the build system with a directory named with the hash's first two characters.

Setting up a package feed

An embedded system project seldom has the need to introduce changes to the Yocto build system. Most of the time and effort is spent in application development, followed by a lesser amount in maybe kernel and bootloader development.

As such, a whole system rebuild is probably done very few times. A new project is usually built from a prebuilt shared state cache, and application development work only needs to be done to perform full or incremental builds of a handful of packages.

Once the packages are built, they need to be installed on the target system for testing. Emulated machines are fine for application development, but most hardware-related work needs to be done on embedded hardware.

Getting ready

An option is to manually copy the build binaries to the target's root filesystem, either copying it to the NFS share on the host system the target is mounting its root filesystem from (as explained in the *Configuring network booting for a development setup* recipe earlier) or using any other method like SCP, FTP, or even a microSD card.

This method is also used by IDEs like Eclipse when debugging an application you are working on. However, this method does not scale well when you need to install several packages and dependencies.

The next option would be to copy the packaged binaries (that is, the RPM, deb, or ipk packages) to the target's filesystem and then use the target's package management system to install them. For this to work, your target's filesystem needs to be built with package management tools. Doing this is as easy as adding the package-management feature to your root filesystem; for example, you may add the following line to your project's `conf/local.conf` file:

```
EXTRA_IMAGE_FEATURES += "package-management"
```

So for an RPM package, you will copy it to the target and use the `rpm` or `smart` utilities to install it. The smart package management tool is GPL licensed and can work with a variety of package formats.

However, the most optimized way to do this is to convert your host system package's output directory into a package feed. For example, if you are using the default RPM package format, you may convert `tmp/deploy/rpm` in your build directory into a package feed that your target can use to update.

For this to work, you need to configure an HTTP server on your computer that serves the packages.

Versioning packages

You also need to make sure that the generated packages are correctly versioned, and that means updating the recipe revision, **PR**, with every change. It is possible to do this manually, but the recommended—and compulsory way if you want to use package feeds—is to use a PR server.

However, the PR server is not enabled by default. The packages generated without a PR server are consistent with each other but offer no update guarantees for a system that is already running.

The simplest PR server configuration is to run it locally on your host system. To do this, you add the following to your `conf/local.conf` file:

```
PRSERV_HOST = "localhost:0"
```

With this setup, update coherency is guaranteed for your feed.

If you want to share your feed with other developers, or you are configuring a build server or package server, you would run a single instance of the PR server by running the following command:

```
$ bitbake-prserv --host <server_ip> --port <port> --start
```

And you will update the project's build configuration to use the centralized PR server, editing `conf/local.conf` as follows:

```
PRSERV_HOST = "<server_ip>:<port>"
```

Also, if you are using a shared state cache as described before, all of the contributors to the shared state cache need to use the same PR server.

Once the feed's integrity is guaranteed, we need to configure an HTTP server to serve the feed.

How to do it...

We will use `lighttpd` for this example, as it is lightweight and easy to configure. Follow these steps:

1. Install the web server:

```
$ sudo apt-get install lighttpd
```

2. By default, the document root specified in the /etc/lighttpd/lighttpd.conf configuration file is /var/www/, so we only need a symlink to our package feed:

```
$ sudo mkdir /var/www/wandboard-quad  
$ sudo ln -s /opt/yocto/fsl-community-bsp/wandboard-  
quad/tmp/deploy/rpm /var/www/wandboard-quad/rpm
```

Next, reload the configuration as follows:

```
$ sudo service lighttpd reload
```

3. Refresh the package index. This needs to be done manually to update the package feed after every build:

```
$ bitbake package-index
```

4. Then we need to configure our target filesystem with the new package feeds:

```
# smart channel --add all type=rpm-md \  
baseurl=http://<server_ip>/wandboard-quad/rpm/all
```

```
# smart channel --add wandboard_quad type=rpm-md \  
baseurl=http://<server_ip>/wandboard-quad/rpm/wandboard_quad
```

```
# smart channel --add cortexa9hf_vfp_neon type=rpm-md \  
baseurl=http://<server_ip>/wandboard-  
quad/rpm/cortexa9hf_vfp_neon
```

5. Once the setup is ready, we will be able to query and update packages from the target's root filesystem with the following:

```
# smart update  
# smart query <package_name>  
# smart install <package_name>
```

To make this change persistent in the target's root filesystem, we can configure the package feeds at compilation time by using the PACKAGE_FEED_URIS variable in conf/local.conf as follows:

```
PACKAGE_FEED_URIS = "http://<server_ip>/wandboard-quad"
```

See also

- ▶ More information and a user manual for the smart utility can be found at <https://labix.org/smart/>

Using build history

When maintaining software for an embedded product, you need a way to know what has changed and how it is going to affect your product.

On a Yocto system, you may need to update a package revision (for instance, to fix a security vulnerability), and you need to make sure what the implications of this change are; for example, in terms of package dependencies and changes to the root filesystem.

Build history enables you to do just that, and we will explore it in this recipe.

How to do it...

To enable build history, add the following to your `conf/local.conf` file:

```
INHERIT += "buildhistory"
```

The following enables information gathering, including dependency graphs:

```
BUILDHISTORY_COMMIT = "1"
```

The preceding line of code enables the storage of build history in a local Git repository.

The Git repository location can be set by the `BUILDHISTORY_DIR` variable, which by default is set to a `buildhistory` directory on your `build` directory.

By default, `buildhistory` tracks changes to packages, images, and SDKs. This is configurable using the `BUILDHISTORY_FEATURES` variable. For example, to track only image changes, add the following to your `conf/local.conf`:

```
BUILDHISTORY_FEATURES = "image"
```

It can also track specific files and copy them to the `buildhistory` directory. By default, this includes only `/etc/passwd` and `/etc/groups`, but it can be used to track any important files like security certificates. The files need to be added with the `BUILDHISTORY_IMAGE_FILES` variable in your `conf/local.conf` file as follows:

```
BUILDHISTORY_IMAGE_FILES += "/path/to/file"
```

Build history will slow down the build, increase the build size, and may also grow the Git directory to an unmanageable size. The recommendation is to enable it on a build server for software releases, or in specific cases, such as when updating production software.

How it works...

When enabled, it will keep a record of the changes to each package and image in the form of a Git repository in a way that can be explored and analyzed.

For a package, it records the following information:

- ▶ Package and recipe revision
- ▶ Dependencies
- ▶ Package size
- ▶ Files

For an image, it records the following information:

- ▶ Build configuration
- ▶ Dependency graphs
- ▶ A list of files that includes ownership and permissions
- ▶ List of installed packages

And for an SDK, it records the following information:

- ▶ SDK configuration
- ▶ List of both host and target files, including ownership and permissions
- ▶ Dependency graphs
- ▶ A list of installed packages

Looking at the build history

Inspecting the Git directory with the build history can be done in several ways:

- ▶ Using Git tools like `gitk` or `git log`.
- ▶ Using the **buildhistory-diff** command-line tool, which displays the differences in a human-readable format.
- ▶ Using a Django-1.4-based web interface. You will need to import the build history data to the application's database after every build. The details are available at <http://git.yoctoproject.org/cgit/cgit.cgi/buildhistory-web/tree/README>.

There's more...

To maintain the build history, it's important to optimize it and avoid it from growing over time. Periodic backups of the build history and clean-ups of older data are important to keep the build history repository at a manageable size.

Once the `buildhistory` directory has been backed up, the following process will trim it and keep only the most recent history:

1. Copy your repository to a temporary RAM filesystem (`tmpfs`) to speed things up. Check the output of the `df -h` command to see which directories are `tmpfs` filesystems and how much space they have available, and use one. For example, in Ubuntu, the `/run/shm` directory is available.
2. Add a graft point for a commit one month ago with no parents:

```
$ git rev-parse "HEAD@{1 month ago}" > .git/info/grafts
```

3. Make the graft point permanent:

```
$ git filter-branch
```

4. Clone a new repository to clean up the remaining Git objects:

```
$ git clone file:///${tmpfs}/buildhistory buildhistory.new
```

5. Replace the old `buildhistory` directory with the new cleaned one:

```
$ rm -rf buildhistory
```

```
$ mv buildhistory.new buildhistory
```

Working with build statistics

The build system can collect build information per task and image. The data may be used to identify areas of optimization of build times and bottlenecks, especially when new recipes are added to the system. This recipe will explain how the build statistics work.

How to do it...

To enable the collection of statistics, your project needs to inherit the `buildstats` class by adding it to `USER_CLASSES` in your `conf/local.conf` file. By default, the `fsl-community-bsp` build project is configured to enable them.

```
USER_CLASSES ?= "buildstats"
```

You can configure the location of these statistics with the `BUILDSTATS_BASE` variable, and by default it is set to the `buildstats` folder in the `tmp` directory under the `build` directory (`tmp/buildstats`).

The `buildstats` folder contains a folder per image with the build stats under a `timestamp` folder. Under it will be a subdirectory per package in your built image, and a `build_stats` file that contains:

- ▶ Host system information
- ▶ Root filesystem location and size
- ▶ Build time
- ▶ Average CPU usage
- ▶ Disk statistics

How it works...

The accuracy of the data depends on the download directory, `DL_DIR`, and the shared state cache directory, `SSTATE_DIR`, existing on the same partition or volume, so you may need to configure them accordingly if you are planning to use the build data.

An example `build-stats` file looks like the following:

```
Host Info: Linux agonzal 3.13.0-35-generic #62-Ubuntu SMP Fri Aug  
15 01:58:42 UTC 2014 x86_64 x86_64  
Build Started: 1411486841.52  
Uncompressed Rootfs size: 6.2M /opt/yocto/fsl-community-  
bsp/wandboard-quad/tmp/work/wandboard_quad-poky-linux-  
gnueabi/core-image-minimal/1.0-r0/rootfs  
Elapsed time: 2878.26 seconds  
CPU usage: 51.5%  
EndIOInProgress: 0  
EndReadsComp: 0  
EndReadsMerged: 55289561  
EndSectRead: 65147300  
EndSectWrite: 250044353  
EndTimeIO: 14415452  
EndTimeReads: 10338443  
EndTimeWrite: 750935284  
EndWTimeIO: 816314180  
EndWritesComp: 0  
StartIOInProgress: 0  
StartReadsComp: 0  
StartReadsMerged: 52319544  
StartSectRead: 59228240  
StartSectWrite: 207536552  
StartTimeIO: 13116200  
StartTimeReads: 8831854  
StartTimeWrite: 3861639688  
StartWTimeIO: 3921064032  
StartWritesComp: 0
```

These disk statistics come from the Linux kernel disk I/O stats (<https://www.kernel.org/doc/Documentation/iostats.txt>). The different elements are explained here:

- ▶ `ReadsComp`: This is the total number of reads completed
- ▶ `ReadsMerged`: This is the total number of adjacent reads merged
- ▶ `SectRead`: This is the total number of sectors read
- ▶ `TimeReads`: This is the total number of milliseconds spent reading
- ▶ `WritesComp`: This is the total number of writes completed
- ▶ `SectWrite`: This is the total number of sectors written
- ▶ `TimeWrite`: This is the total number of milliseconds spent writing

`IOInProgress`: This is the total number of I/Os in progress when reading `/proc/diskstats`

- ▶ `TimeIO`: This is the total number of milliseconds spent performing I/O
- ▶ `WTimeIO`: This is the total number of weighted time while performing I/O

And inside each package, we have a list of tasks; for example, for `ncurses-5.9-r15.1`, we have the following tasks:

- ▶ `do_compile`
- ▶ `do_fetch`
- ▶ `do_package`
- ▶ `do_package_write_rpm`
- ▶ `do_populate_lic`
- ▶ `do_rm_work`
- ▶ `do_configure`
- ▶ `do_install`
- ▶ `do_packagedata`
- ▶ `do_patch`
- ▶ `do_populate_sysroot`
- ▶ `do_unpack`

Each one of them contain, in the same format as earlier, the following:

- ▶ Build time
- ▶ CPU usage
- ▶ Disk stats

There's more...

You can also obtain a graphical representation of the data using the `pybootchartgui.py` tool included in the Poky source. From your project's `build` folder, you can execute the following command to obtain a `bootchart.png` graphic in `/tmp`:

```
$ ./sources/poky/scripts/pybootchartgui/pybootchartgui.py  
tmp/buildstats/core-image-minimal-wandboard-quad/ -o /tmp
```

Debugging the build system

In the last recipe of this chapter, we will explore the different methods available to debug problems with the build system and its metadata.

Getting ready

Let's first introduce some of the usual use cases on a debugging session.

Finding recipes

A good way to check whether a specific package is supported in your current layers is to search for it as follows:

```
$ find -name "*busybox*"
```

This will recursively search all layers for the BusyBox pattern. You can limit the search to recipes and append files by executing:

```
$ find -name "*busybox*.bb*"
```

Dumping BitBake's environment

When developing or debugging package or image recipes, it is very common to ask BitBake to list its environment both globally and for a specific target, be it a package or image.

To dump the global environment and grep for a variable of interest (for example, `DISTRO_FEATURES`), use the following command:

```
$ bitbake -e | grep -w DISTRO_FEATURES
```

Optionally, to locate the source directory for a specific package recipe like BusyBox, use the following command:

```
$ bitbake -e busybox | grep ^S=
```

You could also execute the following command to locate the working directory for a package or image recipe:

```
$ bitbake -e <target> | grep ^WORKDIR=
```

Using the development shell

BitBake offers the `devshell` task to help developers. It is executed with the following command:

```
$ bitbake -c devshell <target>
```

It will unpack and patch the source, and open a new terminal (it will autodetect your terminal type or it can be set with `OE_TERMINAL`) in the target source directory, which has the environment correctly setup.



While in a graphical environment, `devshell` opens a new terminal or console window, but if we are working on a non-graphical environment, like telnet or SSH, you may need to specify `screen` as your terminal in your `conf/local.conf` configuration file as follows:

```
OE_TERMINAL = "screen"
```

Inside the `devshell`, you can run development commands like `configure` and `make` or invoke the cross-compiler directly (use the `$CC` environment variable, which has been set up already).

How to do it...

The starting point for debugging a package build error is the BitBake error message printed on the build process. This will usually point us to the task that failed to build.

To list all the tasks available for a given recipe, with descriptions, we execute the following:

```
$ bitbake -c listtasks <target>
```

If you need to recreate the error, you can force a build with the following:

```
$ bitbake -f <target>
```

Or you can ask BitBake to force-run only a specific task using the following command:

```
$ bitbake -c compile -f <target>
```

Task log and run files

To debug the build errors, BitBake creates two types of useful files per shell task and stores them in a `temp` folder in the working directory. Taking BusyBox as an example, we would look into:

```
/opt/yocto/fsl-community-bsp/wandboard-quad/tmp/work/cortexa9hf-vfp-neon-poky-linux-gnueabi/busybox/1.22.1-r32/temp
```

And find a list of log and run files. The filename format is

`log.do_<task>.<pid>`

and `run.do_<task>.<pid>`.

But luckily, we also have symbolic links, without the `pid` part, that link to the latest version.

The log files will contain the output of the task, and that is usually the only information we need to debug the problem. The run file contains the actual code executed by BitBake to generate the log mentioned before. This is only needed when debugging complex build issues.

Python tasks, on the other hand, do not currently write files as described previously, although it is planned to do so in the future. Python tasks execute internally and log information to the terminal.

Adding logging to recipes

BitBake recipes accept either bash or Python code. Python logging is done through the `bb` class and uses the standard logging Python library module. It has the following components:

- ▶ `bb.plain`: This uses `logger/plain`. It can be used for debugging, but should not be committed to the source.
- ▶ `bb.note`: This uses `logger/info`.
- ▶ `bb.warn`: This uses `logger/warn`.
- ▶ `bb.error`: This uses `logger/error`.
- ▶ `bb.fatal`: This uses `logger/critical` and exits BitBake.
- ▶ `bb.debug`: This should be passed log level as the first argument and uses `logger/debug`.

To print debug output from bash in our recipes, we need to use the `logging` class by executing:

```
inherit logging
```

The `logging` class is inherited by default by all recipes containing `base.bbclass`, so we don't usually have to inherit it explicitly. We will then have access to the following bash functions, which will output to the log files (not to the console) in the `temp` directory inside the working directory as described previously:

- ▶ `bbplain`: This function outputs literally what's passed in. It can be used in debugging but should not be committed to a recipe source.
- ▶ `bbnote`: This function prints with the NOTE prefix.
- ▶ `bbwarn`: This prints a non-fatal warning with the WARNING prefix.
- ▶ `bberror`: This prints a non-fatal error with the ERROR prefix.

- ▶ `bbfatal`: This function halts the build and prints an error message as with `bberror`.
- ▶ `bbdebug`: This function prints debug messages with log level passed as the first argument. It is used with the following format:
`bbdebug [123] "message"`



The bash functions mentioned here do not log to the console but only to the log files.



Looking at dependencies

You can ask BitBake to print the current and provided versions of packages with the following command:

```
$ bitbake --show-versions
```

Another common debugging task is the removal of unwanted dependencies.

To see an overview of pulled-in dependencies, you can use BitBake's verbose output by running this:

```
$ bitbake -v <target>
```

To analyze what dependencies are pulled in by a package, we can ask BitBake to create DOT files that describe these dependencies by running the following:

```
$ bitbake -g <target>
```

The DOT format is a text description language for graphics that is understood by the **GraphViz** open source package and all the utilities that use it. DOT files can be visualized or further processed.

You can omit dependencies from the graph to produce more readable output. For example, to omit dependencies from `glibc`, you would run the following command:

```
$ bitbake -g <target> -I glibc
```

Once the preceding commands have been run, we get three files in the current directory:

- ▶ `package-depends.dot`: This file shows the dependencies between runtime targets
- ▶ `pn-depends.dot`: This file shows the dependencies between recipes
- ▶ `task-depends.dot`: This file shows the dependencies between tasks

There is also a `pn-buildlist` file with a list of packages that would be built by the given target.

The Build System

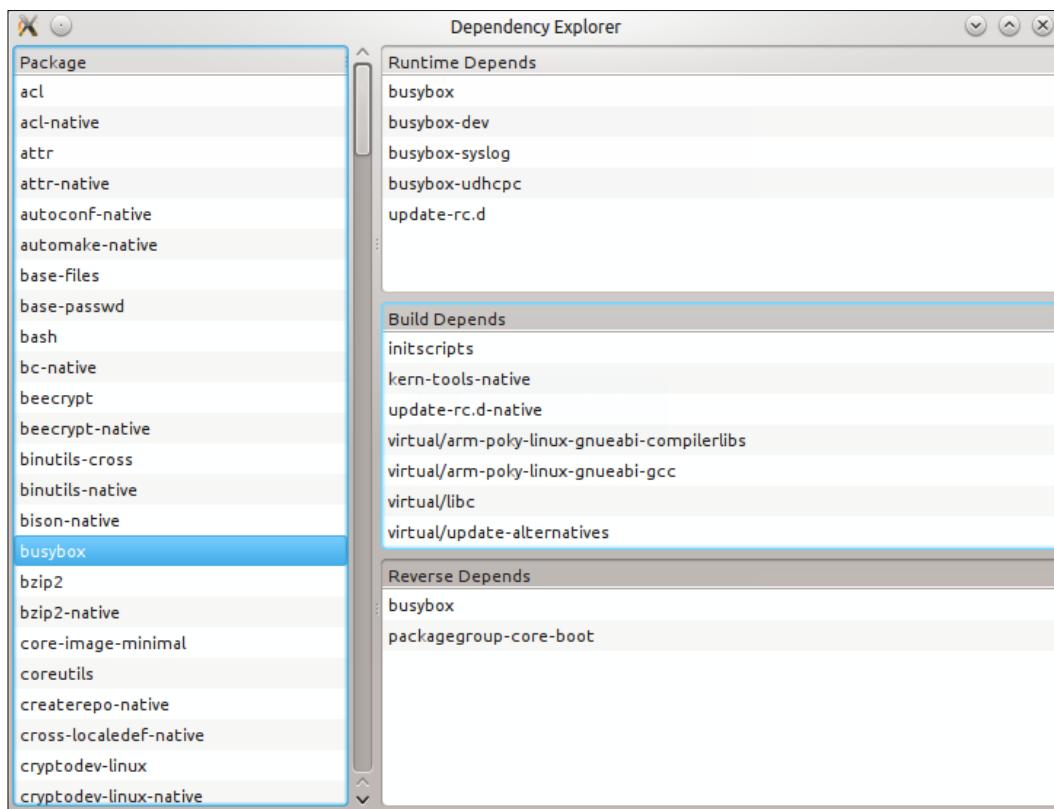
To convert the .dot files to postscript files (.ps), you may execute:

```
$ dot -Tps filename.dot -o outfile.ps
```

However, the most useful way to display dependency data is to ask BitBake to display it graphically with the dependency explorer, as follows:

```
$ bitbake -g -u depexp <target>
```

The result may be seen in the following screenshot:



Debugging BitBake

It is not common to have to debug BitBake itself, but you may find a bug in BitBake and want to explore it by yourself before reporting it to the BitBake community. For such cases, you can ask BitBake to output the debug information at three different levels with the -D flag. To display all the debug information, run the following command:

```
$ bitbake -DDD <target>
```

Error reporting tool

Sometimes, you will find a build error on a Yocto recipe that you have not modified. The first place to check for errors is the community itself, but before launching your mail client, head to <http://errors.yoctoproject.org>.

This is a central database of user-reported errors. Here, you may check whether someone else is experiencing the same problem.

You can submit your own build failure to the database to help the community debug the problem. To do so, you may use the `report-error` class. Add the following to your `conf/local.conf` file:

```
INHERIT += "report-error"
```

By default, the error information is stored under `tmp/log/error-report` under the build directory, but you can set a specific location with the `ERR_REPORT_DIR` variable.

When the error reporting tool is activated, a build error will be captured in a file in the `error-report` folder. The build output will also print a command to send the error log to the server:

```
$ send-error-report ${LOG_DIR}/error-report/error-report_${TSTAMP}
```

When this command is executed, it will report back with a link to the upstream error.

You can set up a local error server, and use that instead by passing a server argument. The error server code and setting up details can be found at <http://git.yoctoproject.org/cgit/cgit.cgi/error-report-web/tree/README>.

There's more...

Although you can use Linux utilities to parse Yocto's metadata and build output, BitBake lacks a command base UI for common tasks. One project that aims to provide it is `bb`, which is available at <https://github.com/kergoth/bb>.

To use it, you need to clone the repository locally by executing the following command:

```
$ cd /opt/yocto/fsl-community-bsp/sources  
$ git clone https://github.com/kergoth/bb.git
```

Then run the `bb/bin/bb init` command, which prompts you to add a bash command to your `~/.bash_profile` file.

You can either do that or execute it in your current shell as follows:

```
$ eval "$(./opt/yocto/fsl-community-bsp/sources/bb/bin/bb init -)"
```

You will first need to set up your environment as usual:

```
$ cd /opt/yocto/fsl-community-bsp  
$ source setup-environment wandboard-quad
```



Some of the commands only work with a populated work directory, so you may need to remove the `rm_work` class if you want to use `bb`.



Some of the tasks that are made easier by the `bb` utility are:

- ▶ Exploring the contents of a package:

```
$ bb contents <target>
```

- ▶ Searching for a pattern in the recipes:

```
$ bb search <pattern>
```

- ▶ Displaying either the global BitBake environment or the environment for a specific package and grepping for a specific variable:

```
$ bb show -r <recipe> <variable>
```

2

The BSP Layer

In this chapter, we will cover the following recipes:

- ▶ Creating a custom BSP layer
- ▶ Introducing system development workflows
- ▶ Adding a custom kernel and bootloader
- ▶ Explaining Yocto's Linux kernel support
- ▶ Describing Linux's build system
- ▶ Configuring the Linux kernel
- ▶ Building the Linux source
- ▶ Building external kernel modules
- ▶ Debugging the Linux kernel and modules
- ▶ Debugging the Linux kernel booting process
- ▶ Using the kernel tracing system
- ▶ Managing the device tree
- ▶ Debugging device tree issues

Introduction

Once we have our build environment ready with the Yocto project, it's time to think about beginning development work on our embedded Linux project.

Most of the embedded Linux projects require both custom hardware and software. An early task in the development process is to test different hardware reference boards and the selection of one to base our design on. We have chosen the Wandboard, a Freescale i.MX6-based platform, as it is an affordable and open board, which makes it perfect for our needs.

On an embedded project, it is usually a good idea to start working on the software as soon as possible, probably before the hardware prototypes are ready, so that it is possible to start working directly with the reference design.

But at some point, the hardware prototypes will be ready and changes will need to be introduced into Yocto to support the new hardware.

This chapter will explain how to create a BSP layer to contain those hardware-specific changes, as well as show how to work with the U-Boot bootloader and the Linux kernel, components which are likely to take most of the customization work.

Creating a custom BSP layer

These custom changes are kept on a separate Yocto layer, called a **Board Support Package (BSP)** layer. This separation is best for future updates and patches to the system. A BSP layer can support any number of new machines and any new software feature that is linked to the hardware itself.

How to do it...

By convention, Yocto layer names start with `meta`, short for metadata. A BSP layer may then add a `bsp` keyword, and finally a unique name. We will call our layer `meta-bsp-custom`.

There are several ways to create a new layer:

- ▶ Manually, once you know what is required
- ▶ By copying the `meta-skeleton` layer included in Poky
- ▶ By using the `yocto-layer` command-line tool

You can have a look at the `meta-skeleton` layer in Poky and see that it includes the following elements:

- ▶ A `layer.conf` file, where the layer configuration variables are set
- ▶ A `COPYING.MIT` license file
- ▶ Several directories named with the `recipes` prefix with example recipes for BusyBox, the Linux kernel and an example module, an example service recipe, an example user management recipe, and a multilib example.

How it works...

We will cover some of the use cases that appear in the available examples in the next few recipes, so for our needs, we will use the `yocto-layer` tool, which allows us to create a minimal layer.

Open a new terminal and change to the `fsl-community-bsp` directory. Then set up the environment as follows:

```
$ source setup-environment wandboard-quad
```



Note that once the build directory has been created, the `MACHINE` variable has already been configured in the `conf/local.conf` file and can be omitted from the command line.

Change to the `sources` directory and run:

```
$ yocto-layer create bsp-custom
```

Note that the `yocto-layer` tool will add the `meta` prefix to your layer, so you don't need to. It will prompt a few questions:

- ▶ The layer priority which is used to decide the layer precedence in cases where the same recipe (with the same name) exists in several layers simultaneously. It is also used to decide in what order `bbappend`s are applied if several layers append the same recipe. Leave the default value of 6. This will be stored in the layer's `conf/layer.conf` file as `BBFILE_PRIORITY`.
- ▶ Whether to create example recipes and append files. Let's leave the default `no` for the time being.

Our new layer has the following structure:

```
meta-bsp-custom/
  conf/layer.conf
  COPYING.MIT
  README
```

There's more...

The first thing to do is to add this new layer to your project's `conf/bblayer.conf` file. It is a good idea to add it to your template `conf` directory's `bblayers.conf.sample` file too, so that it is correctly appended when creating new projects. The highlighted line in the following code shows the addition of the layer to the `conf/bblayers.conf` file:

```
LCONF_VERSION = "6"

BBPATH = "${TOPDIR}"
BSPDIR := "${@os.path.abspath(os.path.dirname(dgetVar('FILE',
    True)) + '/../')}"

BBFILES ?= ""
BBLAYERS = " \
    ${BSPDIR}/sources/poky/meta \
    ${BSPDIR}/sources/poky/meta-yocto \
    \
    ${BSPDIR}/sources/meta-openembedded/meta-oe \
    ${BSPDIR}/sources/meta-openembedded/meta-multimedia \
    \
    ${BSPDIR}/sources/meta-fsl-arm \
    ${BSPDIR}/sources/meta-fsl-arm-extra \
    ${BSPDIR}/sources/meta-fsl-demos \
        ${BSPDIR}/sources/meta-bsp-custom \
"
```

Now, BitBake will parse the `bblayers.conf` file and find the `conf/layers.conf` file from your layer. In it, we find the following line:

```
BBFILES += "${LAYERDIR}/recipes-*/*/*.bb \
${LAYERDIR}/recipes-*/*/*.bbappend"
```

It tells BitBake which directories to parse for recipes and append files. You need to make sure your directory and file hierarchy in this new layer matches the given pattern, or you will need to modify it.

BitBake will also find the following:

```
BBPATH .= ":${LAYERDIR}"
```

The BBPATH variable is used to locate the `bbclass` files and the configuration and files included with the `include` and `require` directives. The search finishes with the first match, so it is best to keep filenames unique.

Some other variables we might consider defining in our `conf/layer.conf` file are:

```
LAYERDEPENDS_bsp-custom = "fsl-arm"
LAYERVERSION_bsp-custom = "1"
```

The LAYERDEPENDS literal is a space-separated list of other layers your layer depends on, and the LAYERVERSION literal specifies the version of your layer in case other layers want to add a dependency to a specific version.

The `COPYING.MIT` file specifies the license for the metadata contained in the layer. The Yocto project is licensed under the *MIT* license, which is also compatible with the **General Public License (GPL)**. This license applies only to the metadata, as every package included in your build will have its own license.

The `README` file will need to be modified for your specific layer. It is usual to describe the layer and provide any other layer dependencies and usage instructions.

Adding a new machine

When customizing your BSP, it is usually a good idea to introduce a new machine for your hardware. These are kept under the `conf/machine` directory in your BSP layer. The usual thing to do is to base it on the reference design. For example, `wandboard-quad` has the following machine configuration file:

```
include include/wandboard.inc

SOC_FAMILY = "mx6:mx6q:wandboard"

UBOOT_MACHINE = "wandboard_quad_config"

KERNEL_DEVICETREE = "imx6q-wandboard.dtb"

MACHINE_FEATURES += "bluetooth wifi"

MACHINE_EXTRA_RRECOMMENDS += " \\"
```

```
bcm4329-nvram-config \
bcm4330-nvram-config \
"
```

A machine based on the Wandboard design could define its own machine configuration file, `wandboard-quad-custom.conf`, as follows:

```
include conf/machine/include/wandboard.inc

SOC_FAMILY = "mx6:mx6q:wandboard"

UBOOT_MACHINE = "wandboard_quad_custom_config"

KERNEL_DEVICETREE = "imx6q-wandboard-custom.dtb"

MACHINE_FEATURES += "wifi"
```

The `wandboard.inc` file now resides on a different layer, so in order for BitBake to find it, we need to specify the full path from the `BBPATH` variable in the corresponding layer. This machine defines its own U-Boot configuration file and Linux kernel device tree in addition to defining its own set of machine features.

Adding a custom device tree to the Linux kernel

To add this device tree file to the Linux kernel, we need to add the device tree file to the `arch/arm/boot/dts` directory under the Linux kernel source and also modify the Linux build system's `arch/arm/boot/dts/Makefile` file to build it as follows:

```
dtb-$ (CONFIG_ARCH_MXC) += \
+imx6q-wandboard-custom.dtb \
```

This code uses diff formatting, where the lines with a minus prefix are removed, the ones with a plus sign are added, and the ones without a prefix are left as reference.

Once the patch is prepared, it can be added to the `meta-bsp-custom/recipes-kernel/linux/linux-wandboard-3.10.17/` directory and the Linux kernel recipe appended adding a `meta-bsp-custom/recipes-kernel/linux/linux-wandboard_3.10.17.bbappend` file with the following content:

```
SRC_URI_append = " file://0001-ARM-dts-Add-wandboard-custom-dts-
file.patch"
```

An example patch that adds a custom device tree to the Linux kernel can be found in the source code that accompanies the book.

Adding a custom U-Boot machine

In the same way, the U-Boot source may be patched to add a new custom machine. Bootloader modifications are not as likely to be needed as kernel modifications though, and most custom platforms will leave the bootloader unchanged. The patch would be added to the `meta-bsp-custom/recipes-bsp/u-boot/u-boot-fslc-v2014.10/` directory and the U-Boot recipe appended with a `meta-bsp-custom/recipes-bsp/u-boot/u-boot-fslc_2014.10.bbappend` file with the following content:

```
SRC_URI_append = " file://0001-boards-Add-wandboard-custom.patch"
```

An example patch that adds a custom machine to U-Boot can be found in the source code that accompanies the book.

Adding a custom formfactor file

Custom platforms can also define their own `formfactor` file with information that the build system cannot obtain from other sources, such as defining whether a touchscreen is available or defining the screen orientation. These are defined in the `recipes-bsp/formfactor/` directory in our `meta-bsp-custom` layer. For our new machine, we could define a `meta-bsp-custom/recipes-bsp/formfactor/formfactor_0.0.bbappend` file to include a `formfactor` file as follows:

```
FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}:"
```

And the machine-specific `meta-bsp-custom/recipes-bsp/formfactor/formfactor/wandboard-quadcUSTOM/machconfig` file would be as follows:

```
HAVE_TOUCHSCREEN=1
```

Introducing system development workflows

When customizing the software, there are some system development workflows that are commonly used, and we will introduce them in this recipe.

How to do it...

We will see an overview of the following development workflows:

- ▶ External development
- ▶ Working directory development
- ▶ External source development

They are all used under different scenarios.

How it works...

Let's understand what the use of each of these development workflows is individually.

External development

In this workflow, we don't use the Yocto build system to build our packages, just a Yocto toolchain and the package's own build system.

The resulting source can be integrated into Yocto in the following ways:

- ▶ With a recipe that fetches a released tarball.
- ▶ With a recipe that fetches directly from a source-controlled repository.

External development is usually the preferred method for U-Boot and Linux kernel development, as they can be easily cross-compiled. Third-party packages in Yocto are also developed in this way.

However, third-party packages can be tricky to cross-compile, and that is just what the Yocto build system makes easy. So, if we are not the main developers of the package and we only want to introduce some fixes or modifications, we can use Yocto to help us. The two workflows explained in the following sections use the Yocto build system.

Working directory development

In this workflow, we use the working directory inside the build directory, `tmp/work`. As we know, when Yocto builds a package, it uses the working directory to extract, patch, configure, build, and package the source. We can directly modify the source in this directory and use the Yocto system to build it.

This methodology is commonly used when sporadically debugging third-party packages.

The workflow is as follows:

1. Remove the package's build directory to start from scratch:
`$ bitbake -c cleanall <target>`
2. Tell BitBake to fetch, unpack, and patch the package, but stop there:
`$ bitbake -c patch <target>`
3. Enter the package's source directory and modify the source. Usually, we would create a temporary local Git directory to help us with our development and to extract the patches easily.
`$ bitbake -c devshell <target>`
4. Build it without losing our changes:
`$ bitbake -C compile <target>`

Note the capital C. This instructs BitBake to run the compile task and all the tasks that follow it. This is the same as running:

```
$ bitbake -c compile <target>  
$ bitbake <target>
```

5. Test it by copying the package to a running system and installing it with the target's package management system. When you run your system from an NFS root filesystem, it's as easy as to copy it there and run the following command (assuming the default RPM package format):

```
$ rpm -i <package>
```

Optionally, you can also use a package feed as we saw in the *Setting up a package feed* recipe in *Chapter 1, The Build System*, in which case you would rebuild the index with the following:

```
$ bitbake package-index
```

And then use the smart package management utility on the target to install the package as previously shown.

6. Extract the patches and add them to the recipe's bbappend file.

External source development

In this workflow, we will use the Yocto build system to build an external directory containing the source. This external directory is usually source controlled to help us in our development.

This is the usual methodology to follow for extensive package development once the source has already been integrated with the Yocto build system.

The workflow is as follows:

1. We perform our development on this external-version-controlled directory and commit our changes locally.
2. We configure the Yocto build system to use a directory in our host system to fetch the source from, and optionally also to build in. This guarantees that our changes cannot be lost by any action of the Yocto build system. We will see some examples of this later on.
3. Build it using Yocto:

```
$ bitbake <target>
```
4. Test it by copying the package to a running system and installing it with the target's package management system.
5. Extract the patches and add them to the recipe's bbappend file.

Adding a custom kernel and bootloader

Development in U-Boot and the Linux kernel is usually done externally to Yocto, as they are easy to build using a toolchain, like the one provided by Yocto.

The development work is then integrated into Yocto in one of two ways:

- ▶ With patches added to the kernel and U-Boot bbappend files. This method will build the same source as the reference design board we are using as base, and apply our changes over it.
- ▶ Using a different Git repository, forked from the Linux kernel and U-Boot Git repositories being used by the reference design, and using a bbappend file to point the recipe to it. This way, we can directly commit the changes to the repository and the Yocto build system will build them.

Usually a forked Git repository is only needed when the hardware changes are substantial and the work in the Linux kernel and bootloader is going to be extensive. The recommendation is to start with patches, and only use a forked repository when they become difficult to manage.

Getting Ready

The first question when starting work on the Linux kernel and U-Boot modifications is how do you find which of the several available recipes are being used for your build.

Finding the Linux kernel source

To find the Linux kernel source, we might use several methods. As we are aware we are building for a wandboard-quad machine, the first thing to do is find a machine configuration file:

```
$ cd /opt/yocto/fsl-community-bsp/sources  
$ find -name wandboard-quad.conf  
.meta-fsl-arm-extra/conf/machine/wandboard-quad.conf
```

The machine configuration file above in turn includes a wandboard.inc file:

```
include conf/machine/include/imx-base.inc  
include conf/machine/include/tune-cortexa9.inc  
  
PREFERRED_PROVIDER_virtual/kernel ?= "linux-wandboard"  
PREFERRED_VERSION_linux-wandboard ?= "3.10.17"
```

Here we find a Linux kernel recipe being specified as the preferred provider for virtual/kernel. Virtual packages like this are used when a feature or element is provided by more than one recipe. It allows us to choose which of all those recipes will finally be used. Virtual packages will be further explained in the *Selecting a specific package versions and providers* recipe in Chapter 3, *The Software Layer*.

We could check the actual output from our previous core-image-minimal build:

```
$ find tmp/work -name "*linux-wandboard*"  
tmp/work/wandboard_quad-poky-linux-gnueabi/linux-wandboard
```

As the linux-wanboard directory exists in our work folder, we can be sure the recipe has been used.

We can check what the available Linux recipes are with:

```
$ find -name "*linux*.bb"
```

We have lots of options, but we can use some of our acquired knowledge to filter them out. Let's exclude the poky and meta-openembedded directories, as we know the BSP support is included in the Freescale community BSP layers:

```
$ find -path ./poky -prune -o -path ./meta-openembedded -prune -o -name  
"*linux*.bb"
```

Finally, we can also use the bitbake-layers script included in Poky:

```
$ cd /opt/yocto/fsl-community-bsp/  
$ source setup-environment wandboard-quad  
$ bitbake-layers show-recipes 'linux*'
```

Not all those kernels support the Wandboard machine completely, but they all support Freescale ARM machines, so they are useful for comparisons.

Finding the U-Boot source

If we continue to pull the include chain, we have imx-base.inc, which itself includes fsl-default-providers.inc, where we find:

```
PREFERRED_PROVIDER_u-boot ??= "u-boot-fslc"  
PREFERRED_PROVIDER_virtual/bootloader ??= "u-boot-fslc"
```

So u-boot-fslc is the U-Boot recipe we are looking for.

Developing using a Git repository fork

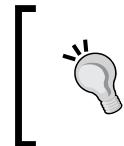
We will show how to append a recipe to use a forked repository to work from it. We will use the Linux kernel as an example, but the concept works just as well for U-Boot or any other package, although the specifics will change.

We will fork or branch the repository used in the reference design and use it to specify SRC_URI for the recipe.

How to do it...

For this example, I have forked the repository to <https://github.com/yoctocookbook/linux>, so my recipes-kernel/linux/linux-wandboard_3.10.17.bbappend file would have the following changes:

```
# Copyright Packt Publishing 2015
WANBOARD_GITHUB_MIRROR = "git://github.com/yoctocookbook/linux.git"
SRCBRANCH = "wandboard_imx_3.10.17_1.0.2_ga-dev"
SRCREV = "${AUTOREV}"
```



Note how the URL needs to start with git://. This is so that BitBake can recognize it as a Git source. Now we can clean and build the Linux kernel and the source will be fetched from the forked repository.

How it works...

Let's have a look at the linux-wandboard_3.10.17.bb recipe:

```
include linux-wandboard.inc
require recipes-kernel/linux/linux-dtb.inc

DEPENDS += "lzop-native bc-native"

# Wandboard branch - based on 3.10.17_1.0.2_ga from Freescale git
SRCBRANCH = "wandboard_imx_3.10.17_1.0.2_ga"
SRCREV = "be8d6872b5eb4c94c15dac36b028ce7f60472409"
LOCALVERSION = "-1.0.2-wandboard"

COMPATIBLE_MACHINE = "(wandboard)"
```

The first interesting thing is the inclusion of both `linux-wandboard.inc` and `linux-dtb.inc`. We will look at the first later on, and the other is a class that allows us to compile Linux kernel device trees. We will discuss device trees in the *Managing the device tree* recipe later in this chapter.

Then it declares two package dependencies, `lzop-native` and `bc-native`. The native part tells us that these are used in the host system, so they are used during the Linux kernel build process. The `lzop` tool is used to create the `cpio` compressed files needed in the `initramfs` system, which is a system that boots from a memory-based root filesystem, and `bc` was introduced to avoid a Perl kernel dependency when generating certain kernel files.

Then it sets the branch and revision, and finally it sets `COMPATIBLE_MACHINE` to `wandboard`. We will speak about machine compatibility in the *Adding new packages* recipe of *Chapter 3, The Software Layer*.

Let's now have a look at the `linux-wandboard.inc` include file:

```
SUMMARY = "Linux kernel for Wandboard"
LICENSE = "GPLv2"
LIC_FILES_CHKSUM =
    "file://COPYING;md5=d7810fab7487fb0aad327b76f1be7cd7"

require recipes-kernel/linux/linux-imx.inc

# Put a local version until we have a true SRCREV to point to
SCMVERSION ?= "y"

SRCBRANCH ??= "master"
LOCALVERSION ?= "-${SRCBRANCH}"

# Allow override of WANBOARD_GITHUB_MIRROR to make use of
# local repository easier
WANBOARD_GITHUB_MIRROR ?= "git://github.com/wandboard-
    org/linux.git"

# SRC_URI for wandboard kernel
SRC_URI = "${WANBOARD_GITHUB_MIRROR};branch=${SRCBRANCH} \
    file://defconfig \
"
```

This is actually the file we were looking for. Initially, it specifies the license for the kernel source and points to it, sets a default branch and local version kernel string, and sets up the `SCR_URI` variable, which is the place where the source code is fetched from.

It then offers the `WANBOARD_GITHUB_MIRROR` variable, which we can modify in our `bbappend` file.

So the logical setup would be to create a GitHub account and fork the provided `wandboard-org` Linux repository.

Once the fork is in place, we need to modify the `WANBOARD_GITHUB_MIRROR` variable. But as we saw before, the recipe configures a specific revision and branch. We want to develop here, so we want to change this to a new development branch we have created. Let's call it `wandboard_imx_3.10.17_1.0.2_ga-dev` and set the revision to automatically fetch the newest point in the branch.

Explaining Yocto's Linux kernel support

The Yocto project offers a kernel framework that allows us to work with the Linux kernel in different ways:

- ▶ Fetching the source from a Git repository and applying patches to it. This is the path taken by the Freescale community BSP-supported kernels, as we saw previously.
- ▶ The `linux-yocto` style kernels that generate the kernel source from a set of Git branches and leafs. Specific features are developed in branches, and a leaf is followed for a complete set of features.

In this recipe, we will show how to work with a `linux-yocto` style kernel.

How to do it...

To use a `linux-yocto` style kernel, the kernel recipe inherits the `linux-yocto.inc` file. A Git repository for a `linux-yocto` style kernel contains metadata either in the recipe or inside the kernel Git tree, in branches named with the `meta` prefix.

The `linux-yocto` style kernel recipes are all named `linux-yocto` and follow the upstream kernel development, rooted in the `kernel.org` repository. Once a new Yocto release cycle starts, a recent upstream kernel version is chosen, and the kernel version from the previous Yocto release is maintained. Older versions are updated inline with the **Long Term Support Initiative (LTSI)** releases. There is also a `linux-yocto-dev` package, which always follows the latest upstream kernel development.

Yocto kernels are maintained separately from the upstream kernel sources, and add features and BSPs to cater to embedded system developers.

Although the Freescale community BSP does not include `linux-yocto` style kernels, some other BSP layers do.

Metadata variables that are used to define the build include:

- ▶ **KMACHINE**: This is usually the same as the `MACHINE` variable, but not always. It defines the kernel's machine type.
- ▶ **KBRANCH**: This explicitly sets the kernel branch to build. It is optional.
- ▶ **KBRANCH_DEFAULT**: This is the default value for `KBRANCH`, initially `master`.
- ▶ **KERNEL_FEATURES**: This adds additional metadata that is used to specify configuration and patches. It appears above the defined `KMACHINE` and `KBRANCH`. It is defined in **Series Configuration Control (SCC)** files as described soon.
- ▶ **LINUX_KERNEL_TYPE**: This defaults to `standard`, but may also be `tiny` or `preempt-rt`. It is defined in its own SCC description files, or explicitly defined using the `KTYPE` variable in the SCC files.

How it works...

The metadata included in the Linux kernel manages the configuration and source selection to support multiple BSPs and kernel types. The tools that manage this metadata are built in the `kern-tools` package.

The metadata can be set either in recipes, for small changes or if you are using a kernel repository you do not have access to, or most usually inside the kernel Git repository in `meta` branches. The `meta` branch that is to be used defaults to a `meta` directory in the same repository branch as the sources, but can be specified using the `KMETA` variable in your kernel recipe. If it does not reside in the same branch as the kernel source, it is kept in an orphan branch; that is, a branch with its own history. To create an orphan branch, use the following commands:

```
$ git checkout --orphan meta
$ git rm -rf .
$ git commit --allow-empty -m "Meta branch"
```

Your recipe must then include `SRCREV_meta` to point to the revision of the `meta` branch to use.

The metadata is described in SCC files, which can include a series of commands:

- ▶ `kconf`: This command applies a configuration fragment to the kernel configuration.
- ▶ `patch`: This command applies the specified patch.
- ▶ `define`: This introduces the variable definitions.
- ▶ `include`: This includes another SCC file.
- ▶ `git merge`: This merges the specified branch into the current branch.

- ▶ branch: This creates a new branch relative to the current branch, usually KTYPE or as specified.

SCC files are broadly divided into the following logical groupings:

- ▶ **configuration (cfg)**: This contains one or more kernel configuration fragments and an SCC file to describe them. For example:

```
cfg/spidev.scc:  
    define KFEATURE_DESCRIPTION "Enable SPI device  
support"  
    kconf hardware spidev.cfg  
  
cfg/spidev.cfg:  
    CONFIG_SPI_SPIDEV=y
```

- ▶ **patches**: This contains one or more kernel patches and an SCC file to describe them. For example:

```
patches/fix.scc:  
    patch fix.patch  
  
patches/fix.patch
```

- ▶ **features**: This contains mix configurations and patches to define complex features. It can also include other description files. For example:

```
features/feature.scc:  
    define KFEATURE_DESCRIPTION "Enable feature"  
  
    patch 0001-feature.patch  
  
    include cfg/feature_dependency.scc  
    kconf non-hardware feature.cfg
```

- ▶ **kernel types**: This contains features that define a high-level kernel policy. By default, three kernel types are defined in SCC files:

- **standard**: This is a generic kernel definition policy
- **tiny**: This is a bare minimum kernel definition policy and is independent of the standard type
- **preempt-rt**: This inherits from the standard type to define a real-time kernel where the PREEMPT-RT patches are applied

Other kernel types can be defined by using the KTYPE variable on an SCC file.

- ▶ **Board Support Packages (BSP):** A combination of kernel types and hardware features. BSP types should include KMACHINE for the kernel machine and KARCH for the kernel architecture.

See also

- ▶ Detailed information regarding `linux-yocto` style kernels can be found in the *Yocto Project Linux Kernel Development Manual* at <http://www.yoctoproject.org/docs/1.7.1/kernel-dev/kernel-dev.html>

Describing Linux's build system

The Linux kernel is a monolithic kernel and as such shares the same address space. Although it has the ability to load modules at runtime, the kernel must contain all the symbols the module uses at compilation time. Once the module is loaded, it will share the kernel's address space.

The kernel build system, or **kbuild**, uses conditional compilation to decide which parts of the kernel are compiled. The kernel build system is independent of the Yocto build system.

In this recipe, we will explain how the kernel's build system works.

How to do it...

The kernel configuration is stored in a `.config` text file in the kernel root directory. The kbuild system reads this configuration to build the kernel. The `.config` file is referred to as the kernel configuration file. There are multiple ways to define a kernel configuration file:

- ▶ Manually editing the `.config` file, although this is not recommended.
- ▶ Using one of the user interfaces the kernel offers (type the `make help` command for other options):
 - ❑ `menuconfig`: An ncurses menu-based interface (`make menuconfig`)
 - ❑ `xconfig`: A Qt-based interface (`make xconfig`)
 - ❑ `gconfig`: A GTK-based interface (`make gconfig`)



Note that to build and use these interfaces, your Linux host needs to have the appropriate dependencies.

- ▶ Automatically via a build system such as Yocto.

Each machine also defines a default configuration in the kernel tree. For ARM platforms, these are stored in the `arch/arm/configs` directory. To configure an ARM kernel, that is, to produce a `.config` file from a default configuration, you run:

```
$ make ARCH=arm <platform>_defconfig
```

For example we can build a default configuration for Freescale i.MX6 processors by running:

```
$ make ARCH=arm imx_v6_v7_defconfig
```

How it works...

Kbuild uses `Makefile` and `Kconfig` files to build the kernel source. `Kconfig` files define configuration symbols and attributes, and `Makefile` file match configuration symbols to source files.

The kbuild system options and targets can be seen by running:

```
$ make ARCH=arm help
```

There's more...

In recent kernels, a default configuration contains all the information needed to expand to a full configuration file. It is a minimal kernel configuration file where all dependencies are removed. To create a default configuration file from a current `.config` file, you run:

```
$ make ARCH=arm savedefconfig
```

This creates a `defconfig` file in the current kernel directory. This `make` target can be seen as the opposite of the `<platform>_defconfig` target explained before. The former creates a configuration file from a minimal configuration, and the other expands the minimal configuration into a full configuration file.

Configuring the Linux kernel

In this recipe, we will explain how to configure a Linux kernel using the Yocto build system.

Getting ready

Before configuring the kernel, we need to provide a default configuration for our machine, which is the one the Yocto project uses to configure a kernel. When defining a new machine in your BSP layer, you need to provide a `defconfig` file.

The Wandboard's defconfig file is stored under `sources/meta-fsl-arm-extra/recipes-kernel/linux/linux-wandboard-3.10.17/defconfig`.

This would be the base defconfig file for our custom hardware, so we copy it to our BSP layer:

```
$ cd /opt/yocto/fsl-community-bsp/sources  
$ mkdir -p meta-bsp-custom/recipes-kernel/linux/linux-wandboard-  
3.10.17/  
$ cp meta-fsl-arm-extra/recipes-kernel/linux/linux-wandboard-  
3.10.17/defconfig meta-bsp-custom/recipes-kernel/linux/linux-  
wandboard-3.10.17/
```

We then add it to our kernel using `meta-bsp-custom/recipes-kernel/linux/linux-wandboard_3.10.17.bbappend` as follows:

```
# Copyright Packt Publishing 2015  
FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}-${PV}:"  
SRC_URI_append = " file://defconfig"
```

Kernel configuration changes to your platform can be made directly in this defconfig file.

How to do it...

To create a .config file from the machine defconfig file, execute the following command:

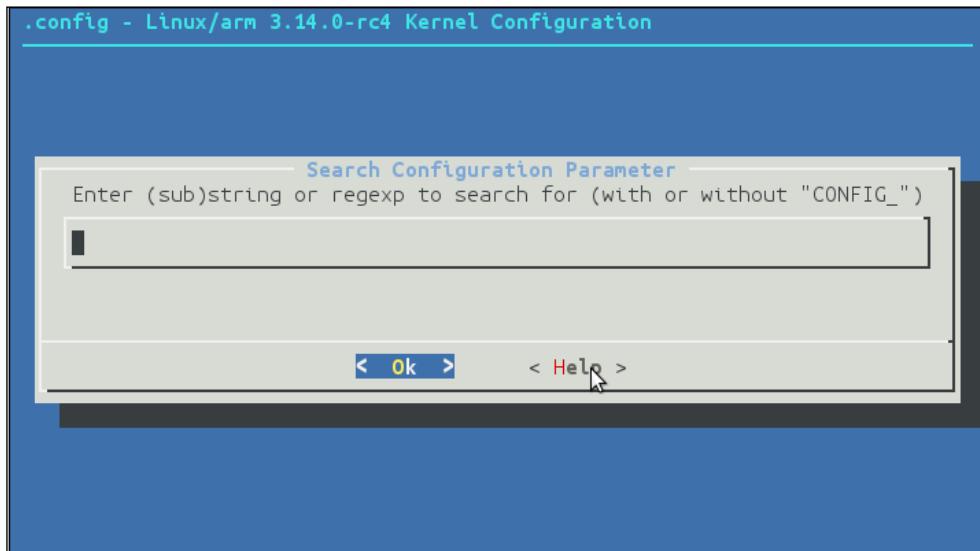
```
$ bitbake -c configure virtual/kernel
```

This will also run the oldconfig kernel make target to validate the configuration against the Linux source.

We can then configure the Linux kernel from the BitBake command line using the following:

```
$ bitbake -c menuconfig virtual/kernel
```

The menuconfig user interface, as well as other kernel configuration user interfaces, has a search functionality that allows you to locate configuration variables by name. Have a look at the following screenshot:



[ In the following chapters, we will mention specific kernel configuration variables, like CONFIG_PRINTK, without specifying the whole path to the configuration variable. The search interface of the different UIs can be used to locate the configuration variable path.]

When you save your changes, a new .config file is created on the kernel's build directory, which you can find using the following command:

```
$ bitbake -e virtual/kernel | grep ^B=
```

You can also modify the configuration using a graphical UI, but not from the BitBake command line. This is because graphical UIs need host dependencies, which are not natively built by Yocto.

To make sure your Ubuntu system has the needed dependencies, execute the following command:

```
$ sudo apt-get install git-core libncurses5 libncurses5-dev libelf-dev asciidoc binutils-dev qt3-dev-tools libqt3-qt4-dev libncurses5 libncurses5-dev fakeroot build-essential crash kexec-tools makedumpfile libgtk2.0-dev libglib2.0-dev libglade2-dev
```

Then change to the kernel build directory, which you found before, with:

```
$ cd /opt/yocto/fsl-community-bsp/wandboard-
quad/tmp/work/wandboard_quad-poky-linux-gnueabi/linux-
wandboard/3.10.17-r0/git
```

Next, run the following:

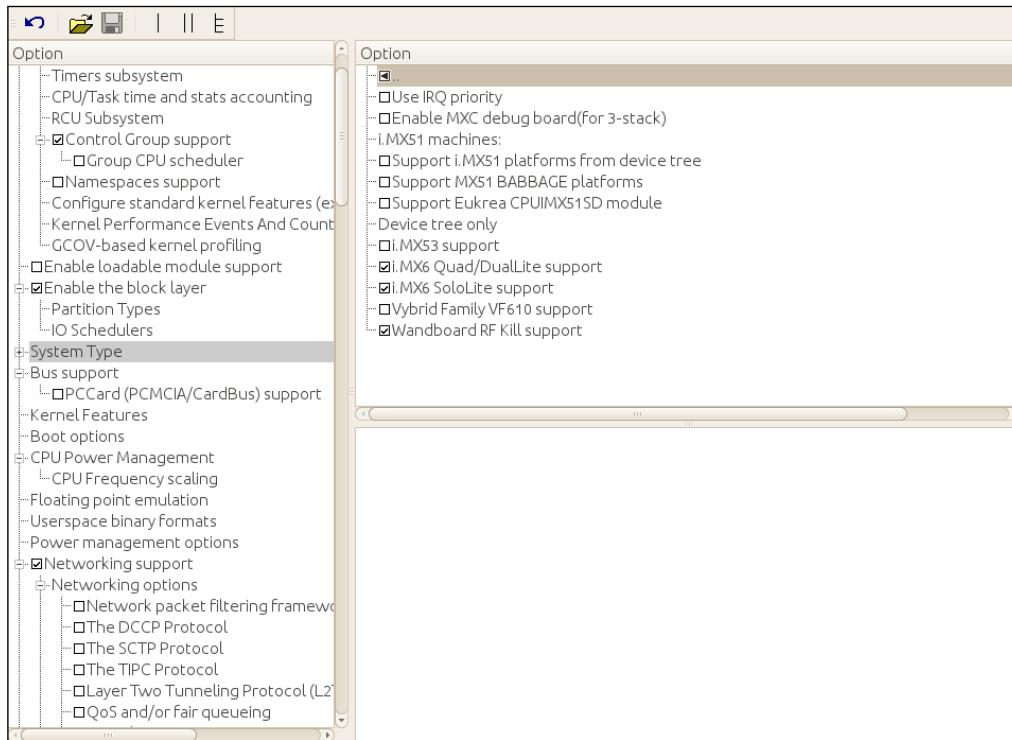
```
$ make ARCH=arm xconfig
```



If you encounter compilation errors, attempt to run from a new terminal that has not had the environment configured with the setup-environment script.



A new window will open with the graphical configuration user interface shown in the next screenshot:



When you save your changes, the `.config` file will be updated.

To use an updated configuration, you need to take care that BitBake does not revert your changes when building. Refer to the *Building the Linux kernel* recipe in this chapter for additional details.

There's more...

You can make your kernel changes permanent with the following steps:

1. Create a default configuration from your `.config` file from the kernel source directory and a clean environment (not configured with the `setup-environment` script) by running:

```
$ make ARCH=arm savedefconfig
```

2. Copy the `defconfig` file from your kernel build folder to your kernel recipe's `defconfig` file as follows:

```
$ cp defconfig /opt/yocto/fsl-community-bsp/sources/meta-bsp-custom/recipes-kernel/linux/linux-wandboard-3.10.17
```

Alternatively, you may use BitBake from the build directory as follows:

```
$ cd /opt/yocto/fsl-community-bsp/  
$ source setup-environment wandboard-quad  
$ bitbake -c savedefconfig virtual/kernel
```

This also creates a `defconfig` file in the Linux kernel's source directory, which needs to be copied to your recipe.

Using configuration fragments

The `linux-yocto` style kernels can also apply isolated kernel configuration changes defined in the kernel configuration fragments; for example:

```
spidev.cfg:  
CONFIG_SPI_SPIDEV=y
```

Kernel configuration fragments are appended to `SRC_URI` in the same way, and are applied over the `defconfig` file.

The `linux-yocto` style kernels (not the one for the Wandboard though) also provide a set of tools to manage kernel configuration:

- ▶ To configure the kernel from the `defconfig` file and the supplied configuration fragments, execute:

```
$ bitbake -f -c kernel_configme linux-yocto
```

- ▶ To create a configuration fragment with your changes, execute:

```
$ bitbake -c diffconfig linux-yocto
```

- ▶ To validate the kernel configuration, you may run:

```
$ bitbake -f -c kernel_configcheck linux-yocto
```

Building the Linux kernel

In this recipe, we will go through the development workflows described earlier using the Linux kernel as an example.

How to do it...

We will see how the following development workflows are applied to the Linux kernel:

- ▶ External development
- ▶ Working directory development
- ▶ External source development

How it works...

Let's explain the three methodologies listed previously in detail.

External development

When compiling outside of the Yocto build environment, we can still use the Yocto-provided toolchain to build. The process is as follows:

1. We will use the Yocto project cross-compilation toolchain already installed in your host.
2. Clone the `wandboard-org linux-wandboard` repository locally:

```
$ cd /opt/yocto  
$ git clone https://github.com/wandboard-org/linux.git linux-wandboard  
$ cd linux-wandboard
```

3. Go to the branch specified in the `linux-wandboard_3.10.17.bb` recipe:

```
$ git checkout -b wandboard_imx_3.10.17_1.0.2_ga  
origin/wandboard_imx_3.10.17_1.0.2_ga
```

4. Compile the kernel source as follows:

- ❑ Prepare the environment as follows:

```
$ source /opt/poky/1.7.1/environment-setup-armv7a-vfp-neon-  
    poky-linux-gnueabi
```

- ❑ Configure the kernel with the default machine configuration:

```
$ cp /opt/yocto/fsl-community-bsp/sources/meta-bsp-custom/  
    recipes-kernel/linux/linux-wandboard-3.10.17/defconfig arch/  
    arm/configs/wandboard-quad_defconfig  
$ make wandboard-quad_defconfig
```

- ❑ Compile the kernel image, modules, and the device tree file with:

```
$ make
```

You can optionally pass a `-jN` argument to `make` to build multithreaded.

This will build the kernel's `zImage`, modules, and device tree files.



Older Yocto environment setup scripts set the `LD` variable to use `gcc`, but the Linux kernel uses `ld` instead. If your compilation is failing, try the following before running `make`:

```
$ unset LDFLAGS
```

To build only modules, you may run:

```
$ make modules
```

And to build only device tree files, you may run:

```
$ make dtbs
```

- ❑ Copy the kernel image and device tree file to the TFTP root to test using network booting:

```
$ cp arch/arm/boot/zImage arch/arm/boot/dts/imx6q-  
    wandboard.dtb /var/lib/tftpboot
```

Some other embedded Linux targets might need to compile a `uImage` if the U-Boot bootloader is not compiled with `zImage` booting support:

```
$ make LOADADDR=0x10800000 uImage
```

 The `mkimage` tool is part of the Yocto toolchain when built with the FSL community BSP. We will see how to build and install an SDK in the *Preparing and using an SDK recipe in Chapter 4, Application Development*.

If it is not included in your toolchain, you can install the tool in your host using the following command:

```
$ sudo apt-get install u-boot-tools
```

`LOADADDR` is the U-Boot entry point; that is, the address where U-Boot will place the kernel in memory. It is defined in the `meta-fsl-arm imx-base.inc` file:

```
UBOOT_ENTRYPOINT_mx6 = "0x10008000"
```

External source development

As we did with U-Boot before, we will use the Yocto build system, pointing it to a local directory with a clone of the Linux source repository. We will use the local Git repository cloned in the earlier section.

We configure for external development in our `conf/local.conf` file using the following code:

```
INHERIT += "externalsrc"
EXTERNALSRC_pn-linux-wandboard = "/opt/yocto/linux-wandboard"
EXTERNALSRC_BUILD_pn-linux-wandboard = "/opt/yocto/linux-
wandboard"
```



Remember to remove this configuration when using the working directory development methodology explained next in this recipe.

But, just as before, the compilation fails with U-Boot. In this case, the `linux-wandboard` recipe, not being a `linux-yocto` style recipe, is not prepared for external source compilation and it fails in the configuration task.

Kernel developers prefer to compile the kernel externally as we saw earlier, so this scenario is not likely to be fixed soon.

Working directory development

Typically we work with patches and use this development workflow when we have a small amount of changes or we don't own the source repository.

A typical workflow when working on a modification would be:

1. Start the kernel package compilation from scratch:

```
$ cd /opt/yocto/fsl-community-bsp/  
$ source setup-environment wandboard-quad  
$ bitbake -c cleanall virtual/kernel
```

This will erase the build folder, shared state cache, and downloaded package source.

2. Configure the kernel as follows:

```
$ bitbake -c configure virtual/kernel
```

This will convert the machine defconfig file into a .config file and call oldconfig to validate the configuration with the kernel source.

You can optionally add your own configuration changes with:

```
$ bitbake -c menuconfig virtual/kernel
```

3. Start a development shell on the kernel:

```
$ bitbake -c devshell virtual/kernel
```

This will fetch, unpack, and patch the kernel sources and spawn a new shell with the environment ready for kernel compilation. The new shell will change to the kernel build directory which contains a local Git repository.

4. Perform our modifications, including kernel configuration changes.
5. Leave the devshell open and go back to the terminal with the sourced Yocto environment to compile the source without erasing our modifications as follows:

```
$ bitbake -C compile virtual/kernel
```

Note the capital c. This invokes the compile task but also all the tasks that follow it.

The newly compiled kernel image is available under tmp/deploy/images/wandboard-quad.

6. Test your changes. Typically, we would work from a network-booted system, so we would copy the kernel image and the device tree file to the TFTP server root and boot the target with them using the following command:

```
$ cd tmp/deploy/images/wandboard-quad/  
$ cp zImage-wandboard-quad.bin zImage-imx6q-wandboard.dtb  
/var/lib/tftpboot
```

Refer to the *Configuring network booting for a development setup* recipe in Chapter 1, *The Build System* for details.

Alternatively, the U-Boot bootloader can boot a Linux zImage kernel from memory with its corresponding device tree using the following syntax:

```
> bootz <kernel_addr> - <dtb_addr>
```

For example, we can fetch images from TFTP and boot the Wandboard images as follows:

```
> tftp ${loadaddr} ${image}
> tftp ${fdt_addr} ${fdt_file}
> bootz ${loadaddr} - ${fdt_addr}
```

If we were using an initramdisk, we would pass it as the second argument. Since we aren't, we use a dash instead.

The command to boot a uImage Linux kernel image from memory would use `bootm` instead, as in:

```
> bootm <kernel_addr> - <dtb_addr>
```

7. Go back to the devshell and commit your change to the local Git repository:

```
$ git add --all .
$ git commit -s -m "Well thought commit message"
```

8. Generate a patch into the kernel recipe patch directory:

```
$ git format-patch -1 -o /opt/yocto/fsl-community-
bsp/sources/meta-fsl-community/recipes-kernel/linux/linux-
wandboard-3.10.17
```

9. Finally, add the patch to the kernel recipe as previously described.

Building external kernel modules

The Linux kernel has the ability to load modules at runtime that extend the kernel functionality. Kernel modules share the kernel's address space and have to be linked against the kernel they are going to be loaded onto. Most device drivers in the Linux kernel can either be compiled into the kernel itself (built-in) or as loadable kernel modules that need to be placed in the root filesystem under the `/lib/modules` directory.

The recommended approach to develop and distribute a kernel module is to do it with the kernel source. A module in the kernel tree uses the kernel's kbuild system to build itself, so as long as it is selected as module in the kernel configuration and the kernel has module support enabled, Yocto will build it.

However, it is not always possible to develop a module in the kernel. Common examples are hardware manufacturers who provide Linux drivers for a wide variety of kernel versions and have an internal development process separated from the kernel community. The internal development work is usually released first as an external out-of-tree module, although it is common for some or all of these internal developments to finish up in the mainstream kernel eventually. However, upstreaming is a slow process and hardware companies will therefore prefer to develop internally first.

It's worth remembering that the Linux kernel is covered under a GPLv2 license, so Linux kernel modules should be released with a compatible license. We will cover licenses in more detail in the following chapters.

Getting ready

To compile an external kernel module with Yocto, we first need to know how we would link the module source with the kernel itself. An external kernel module is also built using the kbuild system of the Linux kernel it is going to be linked against, so the first thing we need is a `Makefile`:

```
obj-m:= hello_world.o

SRC := $(shell pwd)
all:
    $(MAKE) -C $(KERNEL_SRC) M=$(SRC)

modules_install:
    $(MAKE) -C $(KERNEL_SRC) M=$(SRC) modules_install

clean:
    rm -f *.o *~ core .depend .* .cmd *.ko *.mod.c
    rm -f Module.markers Module.symvers modules.order
    rm -rf .tmp_versions Modules.symvers
```

The `Makefile` file just wraps the `make` command used to compile a module on a Linux system:

```
make -C $(KERNEL_SRC) M=$(SRC)
```

Here, `make` is instructed to build in the location of the kernel source, and the `M` argument tells kbuild it is building a module at the specified location.

And then we code the source of the module itself (`hello_world.c`):

```
/ *
 * This program is free software; you can redistribute it and/or
 * modify
```

```
* it under the terms of the GNU General Public License as
published by
* the Free Software Foundation; either version 2 of the License,
or
* (at your option) any later version.
*
* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
* GNU General Public License for more details.
*
* You should have received a copy of the GNU General Public
License
* along with this program. If not, see
<http://www.gnu.org/licenses/>.
*/
```

```
#include <linux/module.h>

static int hello_world_init(void)
{
    printk("Hello world\n");
    return 0;
}

static void hello_world_exit(void)
{
    printk("Bye world\n");
}
module_init(hello_world_init);
module_exit(hello_world_exit);

MODULE_LICENSE("GPL v2");
```

It's worth remembering that we need to compile against a kernel source that has already been built. Use the following steps for compilation:

1. We prepare the environment using the Yocto toolchain environment setup script:

```
$ source /opt/poky/1.7.1/environment-setup-armv7a-vfp-neon-
poky-linux-gnueabi
```
2. Next we build the module. We execute the following from the module source directory:

```
$ KERNEL_SRC=/opt/yocto/linux-wandboard make
```

How to do it...

Once we know how to compile the module externally, we are ready to prepare a Linux kernel module Yocto recipe for it.

We place the module source file and `Makefile` in `recipes-kernel/hello-world/files/` inside our `meta-bsp-custom` layer. We then create a `recipes-kernel/hello-world/hello-world.bb` file with the following content:

```
# Copyright (C) 2015 Packt Publishing.

SUMMARY = "Simplest hello world kernel module."
LICENSE = "GPLv2"
LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/GPL-2.0;md5=801f80980d171dd6425610833a22dbe6"

inherit module

SRC_URI = " \
    file://hello_world.c \
    file://Makefile \
"
S = "${WORKDIR}"

COMPATIBLE_MACHINE = "(wandboard)"
```

The recipe defines the source directory and the two module files after inheriting the `module` class, which takes care of everything. The `KERNEL_SRC` argument in our `Makefile` is set by the module class to `STAGING_KERNEL_DIR`, the location where the kernel class places the Linux kernel headers needed for external module compilation.

We build it with the following command:

```
$ bitbake hello-world
```

The resulting module is called `hello_world.ko`, with the `kernel-module` prefix being added to the package name by the module `bbclass` automatically.

There's more...

The previous instructions will build the module but will not install it in the root filesystem. For that, we need to add a dependency to the root filesystem. This is usually done in machine configuration files using `MACHINE_ESSENTIAL` (for modules that are needed to boot) or `MACHINE_EXTRA` (if they are not essential for boot but needed otherwise), variables.

- ▶ The dependencies that are essential to boot are:
 - `MACHINE_ESSENTIAL_EXTRA_RDEPENDS`: The build will fail if they can't be found
 - `MACHINE_ESSENTIAL_EXTRA_RRECOMMENDS`: The build will not fail if they can't be found
- ▶ The dependencies that are not essential to boot are:
 - `MACHINE_EXTRA_RDEPENDS`: The build will fail if they can't be found
 - `MACHINE_ESSENTIAL_EXTRA_RRECOMMENDS`: The build will not fail if they can't be found

Debugging the Linux kernel and modules

We will highlight some of the most common methods employed by kernel developers to debug kernel issues.

How to do it...

Above all, debugging the Linux kernel remains a manual process, and the most important developer tool is the ability to print debug messages.

The kernel uses the `printf` function, which is very similar syntactically to the `printf` function call from standard C libraries, with the addition of an optional log level. The allowed formats are documented in the kernel source under `Documentation/printk-formats.txt`.

The `printf` functionality needs to be compiled into the kernel with the `CONFIG_PRINTK` configuration variable. You can also configure the Linux kernel to prepend a precise timestamp to every message with the `CONFIG_PRINTK_TIME` configuration variable, or even better, with the `printf.time` kernel command-line argument or through sysfs under `/sys/module/printf/parameters`. Usually all kernels contain `printf` support, and the Wandboard kernel does too, although it is commonly removed on production kernels for small embedded systems.

The `printf` function can be used in any context, interrupt, **non-maskable interrupt (NMI)**, or scheduler. Note that using it inside interrupt context is not recommended.

A useful debug statement to be used during development could be:

```
printf(KERN_INFO "[%s:%d] %pf -> var1: %d var2: %d\n",
      __FUNCTION__, __LINE__, __builtin_return_address(0), var1,
      var2);
```

The first thing to note is that there is no comma between the log level macro and the print format. We then print the function and line where the debug statement is placed and then the parent function. Finally, we print the variables we are actually interested in.

How it works...

The available log levels in `printf` are presented in the following table:

Type	Symbol	Description
Emergency	KERN_EMERG	System is unstable and about to crash
Alert	KERN_ALERT	Immediate action is needed
Critical	KERN_CRIT	Critical software or hardware failure
Error	KERN_ERR	Error condition
Warning	KERN_WARNING	Nothing serious, but might indicate a problem
Notice	KERN_NOTICE	Nothing serious, but user should take note
Information	KERN_INFO	System information
Debug	KERN_DEBUG	Debug messages

If no log level is specified, the default log message as configured in the kernel configuration is used. By default, this is `KERN_WARNING`.

All `printf` statements go to the kernel log buffer, which may wrap around, except debug statements, which only appear if the `DEBUG` symbol is defined. We will see how to enable kernel debug messages soon. The `printf` log buffer must be a power of two, and its size should be set in the `CONFIG_LOG_BUF_SHIFT` kernel configuration variable. You may modify it with the `log_buf_len` kernel command-line parameter.

We print the kernel log buffer with the `dmesg` command. Also, a Yocto user space will have a kernel log daemon running that will log kernel messages to disk under `/var/log/messages`.

Messages above the current console log level will also appear on the console immediately.

The `ignore_loglevel` kernel command-line argument, also available under `/sys/module/printk/parameters`, may be used to print all kernel messages to the console independently of the log level.

You can also change the log level at runtime via the `proc` filesystem. The `/proc/sys/kernel/printk` file contains the current, default, minimum, and boot time default log levels. To change the current log level to the maximum, execute:

```
$ echo 8 > /proc/sys/kernel/printk
```

You can also set the console log level with the `dmesg` tool as follows:

```
$ dmesg -n 8
```

To make the change persistent, you can pass a log level command-line parameter to the kernel, or on some Yocto root filesystem images, you could also use a `/etc/sysctl.conf` file (those that install the `procps` package).

There's more...

Linux drivers do not use the `printk` function directly. They use, in order of preference, subsystem-specific messages (such as `netdev` or `v4l`) or the `dev_*` and `pr_*` family of functions. The latter are described in the following table:

Device message	Generic message	Printk symbol
<code>dev_emerg</code>	<code>pr_emerg</code>	<code>KERN_EMERG</code>
<code>dev_alert</code>	<code>pr_alert</code>	<code>KERN_ALERT</code>
<code>dev_crit</code>	<code>pr_crit</code>	<code>KERN_CRIT</code>
<code>dev_err</code>	<code>pr_err</code>	<code>KERN_ERR</code>
<code>dev_warn</code>	<code>pr_warn</code>	<code>KERN_WARNING</code>
<code>dev_notice</code>	<code>pr_notice</code>	<code>KERN_NOTICE</code>
<code>dev_info</code>	<code>pr_info</code>	<code>KERN_INFO</code>
<code>dev_dbg</code>	<code>pr_debug</code>	<code>KERN_DEBUG</code>

To enable the debug messages within a driver, you may do either of these:

- ▶ Define `DEBUG` in a macro before any other header file in your driver source, as follows:

```
#define DEBUG
```
- ▶ Use the dynamic debug kernel feature. You can then enable/disable all `dev_dbg` and `pr_debug` debug messages with granularity through `debugfs`.

Using dynamic debug

To use the dynamic debug functionality in the Linux kernel, follow these steps:

1. Make sure your kernel is compiled with dynamic debugging (CONFIG_DYNAMIC_DEBUG).

2. Mount the debug filesystem if it hasn't already been mounted:

```
$ mount -t debugfs nodev /sys/kernel/debug
```

3. Configure the debug though the dynamic_debug/control folder. It accepts a whitespace-separated sequence of words:

- ❑ func <function name>
- ❑ file <filename>
- ❑ module <module name>
- ❑ format <pattern>
- ❑ line <line or line range>
- ❑ + <flag>: This adds the specified flag
- ❑ - <flag>: This one removes the specified flag
- ❑ = <flag>: This sets the specified flag

The flags are defined as follows:

- ❑ f: This flag includes the function name in the message
- ❑ l: This flag includes the line number in the message
- ❑ m: This flag includes the module name in the message
- ❑ p: This flag enables the debug message
- ❑ t: This flag includes the thread ID in non-interrupt context messages

4. By default all debug messages are disabled. The control file contains all the available debug points, and by default they have no flags enabled (marked as =_).

5. Now we will enable the debug as follows:

- ❑ Enable all debug statements in a file:

```
echo -n 'file <filename> +p' >  
/sys/kernel/debug/dynamic_debug/control
```

- ❑ Optionally, you could run a specific debug statement:

```
$ echo -n 'file <filename> line nnnn +p' >  
/sys/kernel/debug/dynamic_debug/control
```

6. To list all enabled debug statements, we use the following command:

```
$ awk '$3 != "=_"' /sys/kernel/debug/dynamic_debug/control
```

To make the debug changes persistent, we can pass `dyndbg=<query>` or `module.dyndbg=<query>` to the kernel in the command-line arguments.

Note that the query string needs to be passed surrounded by quotes so that it is correctly parsed. You can concatenate more than one query in the command-line argument by using a semicolon to separate them; for example, `dyndbg="file mxc_v4l2_capture.c +pfl; file ipu_bg_overlay_sdc.c +pfl"`

Rate-limiting debug messages

There are rate-limiting and one-shot extensions to the `dev_*`, `pr_*`, and `printk` family of functions:

- ▶ `printk_ratelimited()`, `pr_*_ratelimited()`, and `dev_*_ratelimited()` print no more than 10 times in a 5 * HZ interval
- ▶ `printk_once()`, `pr_*_once()`, and `dev_*_once()` will print only once.

And you also have utility functions to dump a buffer in hexadecimal; for example, `print_hex_dump_bytes()`.

See also

- ▶ The dynamic debug is documented in the Linux kernel source under `Documentation/dynamic-debug-howto.txt`

Debugging the Linux kernel booting process

We have seen the most general techniques for debugging the Linux kernel. However, some special scenarios require the use of different methods. One of the most common scenarios in embedded Linux development is the debugging of the booting process. This recipe will explain some of the techniques used to debug the kernel's booting process.

How to do it...

A kernel crashing on boot usually provides no output whatsoever on the console. As daunting as that may seem, there are techniques we can use to extract debug information. Early crashes usually happen before the serial console has been initialized, so even if there were log messages, we would not see them. The first thing we will show is how to enable early log messages that do not need the serial driver.

In case that is not enough, we will also show techniques to access the log buffer in memory.

How it works...

Debugging booting problems have two distinctive phases, before and after the serial console is initialized. After the serial is initialized and we can see serial output from the kernel, debugging can use the techniques described earlier.

Before the serial is initialized, however, there is a basic **UART** support in ARM kernels that allows you to use the serial from early boot. This support is compiled in with the `CONFIG_DEBUG_LL` configuration variable.

This adds supports for a debug-only series of assembly functions that allow you to output data to a UART. The low-level support is platform specific, and for the i.MX6, it can be found under `arch/arm/include/debug/imx.S`. The code allows for this low-level UART to be configured through the `CONFIG_DEBUG_IMX_UART_PORT` configuration variable.

We can use this support directly by using the `printascii` function as follows:

```
extern void printascii(const char * );
printascii("Literal string\n");
```

However, much more preferred would be to use the `early_print` function, which makes use of the function explained previously and accepts formatted input in `printf` style; for example:

```
early_print("%08x\t%s\n", p->nr, p->name);
```

Dumping the kernel's printk buffer from the bootloader

Another useful technique to debug Linux kernel crashes at boot is to analyze the kernel log after the crash. This is only possible if the RAM memory is persistent across reboots and does not get initialized by the bootloader.

As U-Boot keeps the memory intact, we can use this method to peek at the kernel log memory in search of clues.

Looking at the kernel source, we can see how the log ring buffer is set up in `kernel/printk/printk.c` and also note that it is stored in `__log_buf`.

To find the location of the kernel buffer, we will use the `System.map` file created by the Linux build process, which maps symbols with virtual addresses using the following command:

```
$grep __log_buf System.map
80f450c0 b __log_buf
```

To convert the virtual address to physical address, we look at how `__virt_to_phys()` is defined for ARM:

```
x - PAGE_OFFSET + PHYS_OFFSET
```

The PAGE_OFFSET variable is defined in the kernel configuration as:

```
config PAGE_OFFSET
    hex
    default 0x40000000 if VMSPLIT_1G
    default 0x80000000 if VMSPLIT_2G
    default 0xC0000000
```

Some of the ARM platforms, like the i.MX6, will dynamically patch the `__virt_to_phys()` translation at runtime, so PHYS_OFFSET will depend on where the kernel is loaded into memory. As this can vary, the calculation we just saw is platform specific.

For the Wandboard, the physical address for 0x80f450c0 is 0x10f450c0.

We can then force a reboot using a magic SysRq key, which needs to be enabled in the kernel configuration with `CONFIG_MAGIC_SYSRQ`, but is enabled in the Wandboard by default:

```
$ echo b > /proc/sysrq-trigger
```

We then dump that memory address from U-Boot as follows:

```
> md.l 0x10f450c0
10f450c0: 00000000 00000000 00210038 c6000000 .....8.!.....
10f450d0: 746f6f42 20676e69 756e694c 6e6f2078 Booting Linux on
10f450e0: 79687020 61636973 5043206c 78302055 physical CPU 0x
10f450f0: 00000030 00000000 00000000 00000000 0.....
10f45100: 009600a8 a6000000 756e694c 65762078 .....Linux ve
10f45110: 6f697372 2e33206e 312e3031 2e312d37 rsion 3.10.17-1.
10f45120: 2d322e30 646e6177 72616f62 62672b64 0.2-wandboard+gb
10f45130: 36643865 62323738 20626535 656c6128 e8d6872b5eb (ale
10f45140: 6f6c4078 696c2d67 2d78756e 612d7068 x@log-linux-hp-a
10f45150: 7a6e6f67 20296c61 63636728 72657620 gonzal) (gcc ver
10f45160: 6e6f6973 392e3420 2820312e 29434347 sion 4.9.1 (GCC)
10f45170: 23202920 4d532031 52502050 504d4545 ) #1 SMP PREEMP
10f45180: 75532054 6546206e 35312062 3a323120 T Sun Feb 15 12:
10f45190: 333a3733 45432037 30322054 00003531 37:37 CET 2015..
10f451a0: 00000000 00000000 00400050 82000000 .....P.@.....
10f451b0: 3a555043 4d524120 50203776 65636f72 CPU: ARMv7 Proce
```

There's more...

Another method is to store the kernel log messages and kernel panics or oops into persistent storage. The Linux kernel's persistent store support (`CONFIG_PSTORE`) allows you to log in to the persistent memory kept across reboots.

To log panic and oops messages into persistent memory, we need to configure the kernel with the `CONFIG_PSTORE_RAM` configuration variable, and to log kernel messages, we need to configure the kernel with `CONFIG_PSTORE_CONSOLE`.

We then need to configure the location of the persistent storage on an unused memory location, but keep the last 1 MB of memory free. For example, we could pass the following kernel command-line arguments to reserve a 128 KB region starting at 0x30000000:

```
ramoops.mem_address=0x30000000 ramoops.mem_size=0x200000
```

We would then mount the persistent storage by adding it to `/etc/fstab` so that it is available on the next boot as well:

```
/etc/fstab:  
pstore /pstore pstore defaults 0 0
```

We then mount it as follows:

```
# mkdir /pstore  
# mount /pstore
```

Next, we force a reboot with the magic `SysRq` key:

```
# echo b > /proc/sysrq-trigger
```

On reboot, we will see a file inside `/pstore`:

```
-r--r--r-- 1 root root 4084 Sep 16 16:24 console-ramoops
```

This will have contents such as the following:

```
SysRq : Resetting  
CPU3: stopping  
CPU: 3 PID: 0 Comm: swapper/3 Not tainted 3.14.0-rc4-1.0.0-wandboard-  
37774-gleae  
[<80014a30>] (unwind_backtrace) from [<800116cc>] (show_stack+0x10/0x14)  
[<800116cc>] (show_stack) from [<806091f4>] (dump_stack+0x7c/0xbc)  
[<806091f4>] (dump_stack) from [<80013990>] (handle_IPI+0x144/0x158)
```

```
[<80013990>] (handle_IPI) from [<800085c4>] (gic_handle_irq+0x58/0x5c)
[<800085c4>] (gic_handle_irq) from [<80012200>] (__irq_svc+0x40/0x70)
Exception stack(0xee4c1f50 to 0xee4c1f98)
```

We should move it out of /pstore or remove it completely so that it doesn't occupy memory.

Using the kernel function tracing system

Recent versions of the Linux kernel contain a set of tracers that, by instrumenting the kernel, allow you to analyze different areas like:

- ▶ Interrupt latency
- ▶ Preemption latency
- ▶ Scheduling latency
- ▶ Process context switches
- ▶ Event tracing
- ▶ Syscalls
- ▶ Maximum stack
- ▶ Block layer
- ▶ Functions

The tracers have no performance overhead when not enabled.

Getting ready...

The tracing system can be used in a wide variety of debugging scenarios, but one of the most common tracers used is the function tracer. It instruments every kernel function with a NOP call that is replaced and used to trace the kernel functions when a trace point is enabled.

To enable the function tracer in the kernel, use the `CONFIG_FUNCTION_TRACER` and `CONFIG_FUNCTION_GRAPH_TRACER` configuration variables.

The kernel tracing system is controlled via a `tracing` file in the debug filesystem, which is mounted by default on Yocto's default images. If not, you can mount it with:

```
$ mount -t debugfs nodev /sys/kernel/debug
```

We can list the available tracers in our kernel by executing:

```
$ cat /sys/kernel/debug/tracing/available_tracers
function_graph function nop
```

How to do it...

You can enable a tracer by echoing its name to the `current_tracer` file. No tracers are enabled by default:

```
$ cat /sys/kernel/debug/tracing/current_tracer
nop
```

You can disable all tracers by executing the following command:

```
$ echo -n nop > /sys/kernel/debug/tracing/current_tracer
```

We use `echo -n` to avoid the trailing newline when echoing to files in `sysfs`.

To enable the function tracer, you would execute:

```
$ echo -n function > /sys/kernel/debug/tracing/current_tracer
```

A prettier graph can be obtained by using the function graph tracer as follows:

```
$ echo -n function_graph > /sys/kernel/debug/tracing/current_tracer
```

How it works...

You can look at the captured trace in human-readable format via the `trace` and `trace_pipe` files, with the latter blocking on `read` and consuming the data.

The function tracer provides the following output:

```
$ cat /sys/kernel/debug/tracing/trace_pipe
root@wandboard-quad:~# cat /sys/kernel/debug/tracing/trace_pipe
        sh-394  [003] ...1    46.205203: mutex_unlock <-
tracing_set_tracer
        sh-394  [003] ...1    46.205215: __fsnotify_parent <-
vfs_write
        sh-394  [003] ...1    46.205218: fsnotify <-vfs_write
        sh-394  [003] ...1    46.205220: __srcu_read_lock <-
fsnotify
        sh-394  [003] ...1    46.205223: preempt_count_add <-
__srcu_read_lock
        sh-394  [003] ...2    46.205226: preempt_count_sub <-
__srcu_read_lock
        sh-394  [003] ...1    46.205229: __srcu_read_unlock <-
fsnotify
```

```
sh-394 [003] ...1 46.205232: __sb_end_write <-
vfs_write
sh-394 [003] ...1 46.205235: preempt_count_add <-
__percpu_counter_add
sh-394 [003] ...2 46.205238: preempt_count_sub <-
__percpu_counter_add
sh-394 [003] d..1 46.205247: gic_handle_irq <-
irq_usr
<idle>-0 [002] d..2 46.205247: ktime_get <-
cpuidle_enter_state
```

The format for the function tracer output is:

```
task-PID [cpu-nr] irqs-off need-resched hard/softirq preempt-depth
delay-timestamp function
```

The graphical function tracer output is as follows:

```
$ cat /sys/kernel/debug/tracing/trace_pipe
3) =====> |
3)           |   gic_handle_irq() {
2) =====> |
2)           |   gic_handle_irq() {
3) 0.637 us |   irq_find_mapping();
2) 0.712 us |   irq_find_mapping();
3)           |   handle_IRQ() {
2)           |   handle_IRQ() {
3)           |   irq_enter() {
2)           |   irq_enter() {
3) 0.652 us |       rcu_irq_enter();
2) 0.666 us |       rcu_irq_enter();
3) 0.591 us |       preempt_count_add();
2) 0.606 us |       preempt_count_add();
```

The format for the graphical function tracer output is:

```
cpu-nr) timestamp | functions
```

There's more...

The kernel tracing system allows us to insert traces in the code by using the `trace_printk` function call. It has the same syntax as `printk` and can be used in the same scenarios, interrupts, NMI, or scheduler contexts.

Its advantage is that as it prints to the tracing buffer in memory and not to the console, it has much lower delays than `printk`, so it is useful to debug scenarios where `printk` is affecting the system's behavior; for example, when masking a timing bug.

Tracing is enabled once a tracer is configured, but whether the trace writes to the ring buffer or not can be controlled. To disable the writing to the buffer, use the following command:

```
$ echo 0 > /sys/kernel/debug/tracing/tracing_on
```

And to re-enable it, use the following command:

```
$ echo 1 > /sys/kernel/debug/tracing/tracing_on
```

You can also enable and disable the tracing from kernel space by using the `tracing_on` and `tracing_off` functions.

Inserted traces will appear in any tracer, including the function tracer, in which case it will appear as a comment.

Filtering function traces

You can get finer granularity in the functions being traced by using the dynamic tracer, which can be enabled with the `CONFIG_DYNAMIC_FTRACE` configuration variable. This is enabled with the tracing functionality by default. This adds two more files, `set_ftrace_filter` and `set_ftrace_notrace`. Adding functions to `set_ftrace_filter` will trace only those functions, and adding them to `set_ftrace_notrace` will not trace them, even if they are also added to `set_ftrace_filter`.

The set of available function names that can be filtered may be obtained by executing the following command:

```
$ cat /sys/kernel/debug/tracing/available_filter_functions
```

Functions can be added with:

```
$ echo -n <function_name> >>
/sys/kernel/debug/tracing/set_ftrace_filter
```

Note that we use the concatenation operator (`>>`) so that the new function is appended to the existing ones.

And functions can also be removed with:

```
$ echo -n '!<function>' >> /sys/kernel/debug/tracing/set_ftrace_filter
```

To remove all functions, just echo a blank line into the file:

```
$ echo > /sys/kernel/debug/tracing/set_ftrace_filter
```

There is a special syntax that adds extra flexibility to the filtering: <function>:<command>:<parameter>

Let's explain each of the components individually:

- ▶ **function**: This specifies the function name. Wildcards are allowed.
- ▶ **command**: This has the following attributes:
 - **mod**: This enables the given function name only in the module specified in the parameter
 - **traceon/traceoff**: This enables or disables tracing when the specified function is hit the numbers of times given in the parameter, or always if no parameter is given.
 - **dump**: Dump the contents of the tracing buffer when the given function is hit.

Here are some examples:

```
$ echo -n 'ipu_*:mod:ipu' >
/sys/kernel/debug/tracing/set_ftrace_filter
$ echo -n 'suspend_enter:dump' >
/sys/kernel/debug/tracing/set_ftrace_filter
$ echo -n 'suspend_enter:traceon' >
/sys/kernel/debug/tracing/set_ftrace_filter
```

Enabling trace options

Traces have a set of options that can be individually enabled in the `/sys/kernel/debug/tracing/options` directory. Some of the most useful options include:

- ▶ **print-parent**: This option displays the caller function too
- ▶ **trace_printk**: This option disables `trace_printk` writing

Using the function tracer on oops

Another alternative to log the kernel messages on oops or panic is to configure the function tracer to dump its buffer contents to the console so that the events leading up to the crash can be analyzed. Use the following command:

```
$ echo 1 > /proc/sys/kernel/ftrace_dump_on_oops
```

The `sysrq-z` combination will also dump the contents of the tracing buffer to the console, as does calling `ftrace_dump()` from the kernel code.

Getting a stack trace for a given function

The tracing code can create a backtrace for every function called. However, this is a dangerous feature and should only be used with a filtered selection of functions. Have a look at the following commands:

```
$ echo -n <function_name> > /sys/kernel/debug/tracing/set_ftrace_filter  
$ echo -n function > /sys/kernel/debug/tracing/current_tracer  
$ echo 1 > /sys/kernel/debug/tracing/options/func_stack_trace  
$ cat /sys/kernel/debug/tracing/trace  
$ echo 0 > /sys/kernel/debug/tracing/options/func_stack_trace  
$ echo > /sys/kernel/debug/tracing/set_ftrace_filter
```

Configuring the function tracer at boot

The function tracer can be configured in the kernel command-line arguments and started as early as possible in the boot process. For example, to configure the graphic function tracer and filter some functions, we would pass the following arguments from the U-Boot bootloader to the kernel:

```
ftrace=function_graph ftrace_filter=mxc_hdmi*,fb_show*
```

See also

- ▶ More details can be found in the kernel source documentation folder at [Documentation/trace/ftrace.txt](#)

Managing the device tree

The device tree is a data structure that is passed to the Linux kernel to describe the physical devices in a system.

In this recipe, we will explain how to work with device trees.

Getting ready

Devices that cannot be discovered by the CPU are handled by the platform devices API on the Linux kernel. The device tree replaces the legacy platform data where hardware characteristics were hardcoded in the kernel source so that platform devices can be instantiated. Before device trees came into use, the bootloader (for example, U-Boot) had to tell the kernel what machine type it was booting. Moreover, it had to pass other information

such as memory size and location, kernel command line, and more.

The device tree should not be confused with the Linux kernel configuration. The device tree specifies what devices are available and how they are accessed, not whether the hardware is used.

The device tree was first used by the PowerPC architecture and was adopted later on by ARM and all others, except x86. It was defined by the Open Firmware specification, which defined the flattened device tree format in **Power.org Standard for Embedded Power Architecture Platform Requirements (ePAPR)**, which describes an interface between a boot program and a client.

Platform customization changes will usually happen in the device tree without the need to modify the kernel source.

How to do it...

A device tree is defined in a human-readable device tree syntax (.dts) text file. Every board has one or several DTS files that correspond to different hardware configurations.

These DTS files are compiled into **Device Tree Binary (DTB)** blobs, which have the following properties:

- ▶ They are relocatable, so pointers are never used internally
- ▶ They allow for dynamic node insertion and removal
- ▶ They are small in size

Device tree blobs can either be attached to the kernel binary (for legacy compatibility) or, as is more commonly done, passed to the kernel by a bootloader like U-Boot.

To compile them, we use a **Device Tree Compiler (DTC)**, which is included in the kernel source inside `scripts/dtc` and is compiled along with the kernel itself, or we could alternatively install it as part of your distribution. It is recommended to use the DTC compiler included in the kernel tree.

The device trees can be compiled independently or with the Linux kernel kbuild system, as we saw previously. However, when compiling independently, modern device trees will need to be preprocessed by the C preprocessor first.

It's important to note that the DTC currently performs syntax checking but no binding checking, so invalid DTS files may be compiled, and the resulting DTB file may result in a non-booting kernel. Invalid DTB files usually hang the Linux kernel very early on so there will be no serial output.

The bootloader might also modify the device tree before passing it to the kernel.

How it works...

The DTS file for the wandboard-quad variant is under `arch/arm/boot/dts/imx6q-wandboard.dts` and looks as follows:

```
#include "imx6q.dtsi"
#include "imx6qdl-wandboard.dtsi"

/ {
    model = "Wandboard i.MX6 Quad Board";
    compatible = "wand,imx6q-wandboard", "fsl,imx6q";

    memory {
        reg = <0x10000000 0x80000000>;
    };
}
```

What we see here is the device tree root node that has no parents. The rest of the nodes will have a parent. The structure of a node can be represented as follows:

```
node@0{
    an-empty-property;
    a-string-property = "a string";
    a-string-list-property = "first string", "second string";
    a-cell-property = <1>;
    a-cell-property = <0x1 0x2>;
    a-byte-data-property = [0x1 0x2 0x3 0x4];
    a-phandle-property = <&node1>;
}
```

The node properties can be:

- ▶ Empty
- ▶ Contain one or more strings
- ▶ Contain one or more unsigned 32-bit numbers, called **cells**
- ▶ Contain a binary byte stream
- ▶ Be a reference to another node, called a **phandle**

The device tree is initially parsed by the C preprocessor and it can include other DTS files.

These `include` files have the same syntax and are usually appended with the `.dtsi` suffix. File inclusion can also be performed with the device tree `/include/` operator, although `#include` is recommended, and they should not be mixed. In this case, both `imx6q.dtsi` and `imx6qdl-wandboard.dtsi` are overlaid with the contents of `imx6q-wandboard.dts`.

Device tree nodes are documented in bindings contained in the Documentation/devicetree/bindings/ directory of the kernel source. New nodes must include the corresponding bindings, and these must be reviewed and accepted by the device tree maintainers. Theoretically, all bindings need to be maintained, although it is likely this will be relaxed in the future.

The compatible property

The most important property in a device tree node is the `compatible` property. In the root node, it defines the machine types the device tree is compatible with. The DTS file we just saw is compatible in order of precedence with the `wand,imx6q-wandboard` and `fsl,imx6q` machine types.

On a non-root node, it will define the driver match for the device tree node, binding a device with the driver. For example, a platform driver that binds with a node that defines a property that is compatible with `fsl,imx6q-tempmon` would contain the following excerpt:

```
static const struct of_device_id of_imx_thermal_match[] = {
    { .compatible = "fsl,imx6q-tempmon", },
    { /* end */ }
};

MODULE_DEVICE_TABLE(of, of_imx_thermal_match);

static struct platform_driver imx_thermal = {
    .driver = {
        .name      = "imx_thermal",
        .owner     = THIS_MODULE,
        .of_match_table = of_imx_thermal_match,
    },
    .probe      = imx_thermal_probe,
    .remove    = imx_thermal_remove,
};
module_platform_driver(imx_thermal);
```

The Wandboard device tree file

Usually, the first DTSI file to be included is `skeleton.dtsi`, which is the minimum device tree needed to boot, once a compatible property is added.

```
/ {
    #address-cells = <1>;
    #size-cells = <1>;
    chosen { };
    aliases { };
    memory { device_type = "memory"; reg = <0 0>; };
};
```

Here are the other common top nodes:

- ▶ **chosen**: This node defines fixed parameters set at boot, such as the Linux kernel command line or the `initramfs` memory location. It replaces the information traditionally passed in ARM tags (ATAGS).
- ▶ **memory**: This node is used to define the location and size of RAM. This is usually filled in by the bootloader.
- ▶ **aliases**: This defines shortcuts to other nodes.
- ▶ **address-cells** and **size-cells**: These are used for memory addressability and will be discussed later on.

A summary representation of the `imx6q-wandboard.dts` file showing only the selected buses and devices follows:

```
#include "skeleton.dtsci"

{
    model = "Wandboard i.MX6 Quad Board";
    compatible = "wand,imx6q-wandboard", "fsl,imx6q";

    memory {};

    aliases {};

    intc: interrupt-controller@00a01000 {};

    soc {
        compatible = "simple-bus";

        dma_apbh: dma-apbh@00110000 {};
        timer@00a00600 {};

        L2: l2-cache@00a02000 {};

        pcie: pcie@0x01000000 {};

        aips-bus@02000000 { /* AIPS1 */
            compatible = "fsl,aips-bus", "simple-bus";

            spba-bus@02000000 {
                compatible = "fsl,spba-bus", "simple-bus";
            }
        }
    }
}
```

```
    };
    aipstz@0207c000 {};
    clks: ccm@020c4000 {};
    iomuxc: iomuxc@020e0000 {};
};

aips-bus@02100000 {
    compatible = "fsl,aips-bus", "simple-bus";
};
};

};
```

On this DTS, we can find several nodes defining **system on chip (SoC)** buses and several other nodes defining on-board devices.

Defining buses and memory-addressable devices

Buses are typically defined by the `compatible` property or the `simple-bus` property (to define a memory-mapped bus with no specific driver binding) or both. The `simple-bus` property is needed so that children nodes to the bus are registered as platform devices.

For example, the `soc` node is defined as follows:

```
soc {
    compatible = "simple-bus";
    #address-cells = <1>;
    #size-cells = <1>;
    ranges;

    aips-bus@02000000 { /* AIPS1 */
        compatible = "fsl,aips-bus", "simple-bus";
        reg = <0x02000000 0x100000>;
    }
}
```

The properties on the `soc` node are used to specify the memory addressability of the children nodes.

- ▶ `address-cells`: This property indicates how many base address cells are needed in the `reg` property.
- ▶ `size-cells`: This property indicates how many size cells are needed in the `reg` property.
- ▶ `ranges`: This one describes an address translation between parent and child buses. In here, there is no translation and parent and child addressing is identical.

In this case, any child of `soc` needs to define its memory addressing with a `reg` property that contains one cell for the address and one cell for the size. The `aips-bus` node does that with the following property:

```
reg = <0x02000000 0x100000>;
```

There's more...

When the device tree binary blob is loaded in memory by the Linux kernel, it is expanded into a flattened device tree that is accessed by offset. The `fdt_*` kernel functions are used to access the flattened device tree. This `fdt` is then parsed and transformed into a tree memory structure that can be efficiently accessed with the `of_*` family of functions (the prefix comes from Open Firmware).

Modifying and compiling the device tree in Yocto

To modify the device tree in the Yocto build system, we execute the following set of commands:

```
$ cd /opt/yocto/fsl-community-bsp/  
$ source setup-environment wandboard-quad  
$ bitbake -c devshell virtual/kernel
```

We then edit `arch/arm/boot/dts/imx6q-wandboard.dts` and compile the changes with:

```
$ make dtbs
```

If we want to create a device tree with extra space, let's say 1024 bytes (for example, to add nodes dynamically as explained in the next recipe), we need to specify it with a DTC flag as follows:

```
DTC_FLAGS="-p 1024" make dtbs
```

To deploy it, we exit the devshell and build the kernel from the project's `build` directory:

```
$ bitbake -c deploy -f virtual/kernel
```

See also

- ▶ More information regarding device trees can be found at <http://www.devicetree.org>.

Debugging device tree issues

This recipe will show some techniques to debug common problems with the device tree.

How to do it...

As mentioned before, problems with the syntax of device tree files usually result in the kernel crashing early in the boot process. Other type of problems are more subtle and usually appear once a driver is making use of the information provided by the device tree. For both types of problems, it is helpful to be able to look not only at the device tree syntax file, but also at the device tree blob, as it is read by both U-Boot and the Linux kernel. It may also be helpful to modify the device tree on the fly using the tools that U-Boot offers.

How it works...

Looking at the device tree from U-Boot

The U-Boot bootloader offers the `fdt` command to interact with a device tree blob. On the Wandboard's default environment, there are two variables related to the device tree:

- ▶ `fdt_file`: This variable contains the name of the device tree file used
- ▶ `fdt_addr`: This variable contains the location in memory to load the device tree

To fetch the Wandboard's device tree from the TFTP server location and place it in memory, we use the following command:

```
> tftp ${fdt_addr} ${fdt_file}
```

Once we have the device tree blob in memory, we tell U-Boot where it is located:

```
> fdt addr ${fdt_addr}
```

And then we can inspect nodes from the device tree using the full path to them from the root node. To inspect the selected levels, we use the `list` command, and to print complete subtrees, we use the `print` command:

```
> fdt list /cpus
cpus {
    #address-cells = <0x00000001>;
    #size-cells = <0x00000000>;
    cpu@0 {
    };
};
```

```
> fdt print /cpus
cpus {
    #address-cells = <0x00000001>;
    #size-cells = <0x00000000>;
    cpu@0 {
        compatible = "arm,cortex-a9";
        device_type = "cpu";
        reg = <0x00000000>;
        next-level-cache = <0x00000001d>;
        [omitted]
    };
};

};
```

U-Boot can also attach new nodes to the tree assuming there is extra space in the device tree.

```
> fdt mknod / new-node  
> fdt list /new-node  
new-node {  
};
```

It can also create or remove properties:

```
> fdt set /new-node testprop testvalue
> fdt print /new-node
new-node {
    testprop = "testvalue";
};

> fdt rm /new-node testprop
> fdt print /new-node
new-node {
};

}
```

For example, it can be useful to modify the kernel command line through the chosen node.

Looking at the device tree from the Linux kernel

Once the Linux kernel is booted, it can be useful to expose the device tree to user space so that it can be explored. You can do this by configuring the Linux kernel with the CONFIG_PROC_DEVICETREE configuration variable. The Wandboard Linux kernel comes preconfigured to expose the device tree in /proc/device-tree as follows:

```
# ls /proc/device-tree/cpus/
#address-cells    cpu@0          cpu@2          name
#size-cells       cpu@1          cpu@3
```


3

The Software Layer

In this chapter, we will cover the following recipes:

- ▶ Exploring an image's contents
- ▶ Adding a new software layer
- ▶ Selecting a specific package versions and providers
- ▶ Adding supported packages
- ▶ Adding new packages
- ▶ Adding data, scripts, or configuration files
- ▶ Managing users and groups
- ▶ Using the sysvinit initialization system
- ▶ Using the systemd initialization system
- ▶ Installing package-installation scripts
- ▶ Reducing the Linux kernel image size
- ▶ Reducing the root filesystem image size
- ▶ Releasing software
- ▶ Analyzing your system for compliance
- ▶ Working with open source and proprietary code

Introduction

With hardware-specific changes on their way, the next step is customizing the target root filesystem; that is, the software that runs under the Linux kernel, also called the Linux user space.

The usual approach to this is to start with one of the available core images and both optimize and customize it as per the needs of your embedded project. Usually, the images chosen as a starting point are either `core-image-minimal` or `core-image-sato`, but any of them will do.

This chapter will show you how to add a software layer to contain those changes, and will explain some of the common customizations made, such as size optimization. It will also show you how to add new packages to your root filesystem, including licensing considerations.

Exploring an image's contents

We have already seen how to use the build history feature to obtain a list of packages and files included in our image. In this recipe, we will explain how the root filesystem is built so that we are able to track its components.

Getting ready

When packages are built, they are classified inside the working directory of your project (`tmp/work`) according to their architecture. For example, on a `wandboard-quad` build, we find the following directories:

- ▶ `all-poky-linux`: This is used for architecture-independent packages
- ▶ `cortexa9hf-vfp-neon-poky-linux-gnueabi`: This is used for cortexa9, hard floating point packages
- ▶ `wandboard_quad-poky-linux-gnueabi`: This is used for machine-specific packages; in this case, `wandboard-quad`
- ▶ `x86_64-linux`: This is used for the packages that form the host `sysroot`

BitBake will build all the packages included in its dependency list inside its own directory.

How to do it...

To find the `build` directory for a given package, we can execute the following command:

```
$ bitbake -e <package> | grep ^WORKDIR=
```

Inside the `build` directory, we find some subdirectories (assuming `rm_work` is not used) that the build system uses in the packaging task. These subdirectories include the following:

- ▶ `deploy-rpms`: This is the directory where the final packages are stored. We look here for individual packages that can be locally copied to a target and installed. These packages are copied to the `tmp/deploy` directory and are also used when Yocto builds the root filesystem image.

- ▶ `image`: This is the default destination directory where the `do_install` task installs components. It can be modified by the recipe with the `D` configuration variable.
- ▶ `package`: This one contains the actual package contents.
- ▶ `package-split`: This is where the contents are categorized in subdirectories named after their final packages. Recipes can split the package contents into several final packages, as specified by the `PACKAGES` variable. The default packages besides the default package name are:
 - `dbg`: This installs components used in debugging
 - `dev`: This installs components used in development, such as headers and libraries
 - `staticdev`: This installs libraries and headers used in static compilation
 - `doc`: This is where the documentation is placed
 - `locale`: This installs localization components

The components to be installed in each package are selected using the `FILES` variable. For example, to add to the default package, you could execute the following command:

```
FILES_${PN} += "${bindir}/file.bin"
```

And to add to the development package, you could use the following:

```
FILES_${PN}-dev += "${libdir}/lib.so"
```

How it works...

Once the Yocto build system has built all the individual packages in its dependency list, it runs the `do_rootfs` task, which populates the `sysroot` and builds the root filesystem before creating the final package images. You can find the location of the root filesystem by executing:

```
$ bitbake -e core-image-minimal | grep ^IMAGE_ROOTFS=
```

Note that the `IMAGE_ROOTFS` variable is not configurable and should not be changed.

The contents of this directory will later be prepared into an image according to what image types are configured in the `IMAGE_FSTYPES` configuration variable. If something has been installed in this directory, it will then be installed in the final image.

Adding a new software layer

Root filesystem customization involves adding or modifying content to the base image. Metadata for this content goes into one or more software layers, depending on the amount of customization needed.

A typical embedded project will have just one software layer containing all non-hardware-specific customizations. But it is also possible to have extra layers for graphical frameworks or system-wide elements.

Getting ready

Before starting work on a new layer, it is good practice to check whether someone else provides a similar layer. Also, if you are trying to integrate an open source project, check whether a layer for it already exists. There is an index of available layers at <http://layers.openembedded.org/>.

How to do it...

We can then create a new `meta-custom` layer using the `yocto-layer` command as we learned in the *Creating a custom BSP layer* recipe in *Chapter 2, The BSP Layer*. From the `sources` directory, execute the following command:

```
$ yocto-layer create custom
```

Don't forget to add the layer to your project's `conf/bblayers.conf` file and to your template's `conf` directory to make it available for all new projects.

The default `conf/layer.conf` configuration file is as follows:

```
# We have a conf and classes directory, add to BBPATH
BBPATH .= ":${LAYERDIR}"

# We have recipes-* directories, add to BBFILES
BBFILES += "${LAYERDIR}/recipes-*/*/*.bb \
           ${LAYERDIR}/recipes-*/*/*.bbappend"

BBFILE_COLLECTIONS += "custom"
BBFILE_PATTERN_custom = "^${LAYERDIR} /"
BBFILE_PRIORITY_custom = "6"
```

We have discussed all the relevant variables in this snippet in the *Creating a custom BSP layer* recipe in *Chapter 2, The BSP Layer*.

How it works...

When adding content to a new software layer, we need to keep in mind that our layer needs to play well with other layers in the Yocto project. To this end, when customizing recipes, we will always use append files, and will only override existing recipes if we are completely sure there is no way to add the customization required through an append file.

To help us manage the content across several layers, we can use the following `bitbake-layers` command-line utilities:

- ▶ `$ bitbake-layers show-layers`: This will display the configured layers as BitBake sees them. It is helpful to detect errors on your `conf/bblayer.conf` file.
- ▶ `$ bitbake-layers show-recipes`: This command will display all the available recipes and the layers that provide them. It can be used to verify that BitBake is seeing your newly created recipe. If it does not appear, verify that the filesystem hierarchy corresponds to the one defined in your layer's `BBFILES` variable in `conf/layer.conf`.
- ▶ `$ bitbake-layers show-overlaid`: This command will show all the recipes that are overlaid by another recipe with the same name but in a higher priority layer. It helps detect recipe clashes.
- ▶ `$ bitbake-layers show-appends`: This command will list all available append files and the recipe files they apply to. It can be used to verify that BitBake is seeing your append files. Also, as before with recipes, if they don't appear, you will need to check the filesystem hierarchy and your layer's `BBFILES` variable.
- ▶ `$ bitbake-layers flatten <output_dir>`: This command will create a directory with the contents of all configured layers without overlayed recipes and with all the append files applied. This is how BitBake will see the metadata. This flattened directory is useful to discover conflicts with your layer's metadata.

There's more...

We will sometimes add customizations that are specific to one board or machine. These are not always hardware-related, so they could be found both in a BSP or software layer.

When doing so, we will try to keep our customizations as specific as possible. One typical example is customizing for a specific machine or machine family. If you need to add a patch for the `wandboard-quad` machine, you would use the following line of code:

```
SRC_URI_append_wandboard-quad = " file://mypatch.patch"
```

And, if the patch is applicable to all i.MX6-based boards, you can use the following:

```
SRC_URI_append_mx6 = " file://mypatch.patch"
```

To be able to use machine families overrides, the machine configuration files need to include a `SOC_FAMILY` variable, such as the one for the `wandboard-quad` in `meta-fsl-arm-extra`. Refer to the following line of code:

```
conf/machine/wandboard-quad.conf:SOC_FAMILY = "mx6:mx6q:wandboard"
```

And for it to appear in the MACHINEOVERRIDES variable, the `soc-family.inc` file needs to be included, as it is in `meta-fsl-arm`. Here is the relevant code excerpt from the `conf/machine/include/imx-base.inc` file:

```
include conf/machine/include/soc-family.inc
MACHINEOVERRIDES =. "${@['', '${SOC_FAMILY}:' ]['${SOC_FAMILY}' != '' ]}"
```

BitBake will search a predefined path, looking for files inside the package's working directory, defined in the FILESPATH variable as a colon-separated list. Specifically:

```
 ${PN}-${PV}/${DISTRO}
 ${PN}/${DISTRO}
 files/${DISTRO}

 ${PN}-${PV}/${MACHINE}
 ${PN}/${MACHINE}
 files/${MACHINE}

 ${PN}-${PV}/${SOC_FAMILY}
 ${PN}/${SOC_FAMILY}
 files/${SOC_FAMILY}

 ${PN}-${PV}/${TARGET_ARCH}
 ${PN}/${TARGET_ARCH}
 files/${TARGET_ARCH}

 ${PN}-${PV}/
 ${PN}/
 files/
```

In the specific case of the `wandboard-quad`, this translates to the following:

```
 ${PN}-${PV}/poky
 ${PN}/poky
 files/poky
 ${PN}-${PV}/wandboard-quad
 ${PN}/wandboard-quad
 files/wandboard-quad
 ${PN}-${PV}/wandboard
 ${PN}/wandboard
 files/wandboard
 ${PN}-${PV}/mx6q
 ${PN}/mx6q
 files/mx6q
 ${PN}-${PV}/mx6
 ${PN}/mx6
```

```
files/mx6
${PN}-$PV/armv7a
${PN}/armv7a
files/armv7a
${PN}-$PV/arm
${PN}/arm
files/arm
${PN}-$PV/
${PN}/
files/
```

Here, `PN` is the package name and `PV` is the package version.

It is best to place patches in the most specific of these, so `wandboard-quad`, followed by `wandboard`, `mx6q`, `mx6`, `armv7a`, `arm`, and finally the generic `PN-PV`, `PN`, and `files`.

Note that the search path refers to the location of the BitBake recipe, so append files need to always add the path when adding content. Our append files can add extra folders to this search path if needed by appending or prepending to the `FILESEXTRAPATHS` variable as follows:

```
FILESEXTRAPATHS_prepend := "${THISDIR}/folder:"
```

 Note the immediate operator (`:=`) that expands `THISDIR` immediately, and the prepend that places your added path before any other path so that your patches and files are found first in the search. Also, we have seen the `+ =` and `= +` style of operators in configuration files, but they should be avoided in recipe files and the append and prepend operators should be given preference, as seen in the example code explained previously to avoid ordering issues.

Selecting a specific package version and providers

Our layers can provide recipes for different versions of the same package. For example, the `meta-fsl-arm` layer contains several different types of Linux sources:

- ▶ `linux-imx`: This corresponds to the Freescale BSP kernel image fetched from <http://git.freescale.com/git/cgit.cgi/imx/linux-2.6-imx.git/>
- ▶ `linux-fslc`: This is the mainline Linux kernel and fetched from <https://github.com/Freescale/linux-fslc>
- ▶ `linux-timesys`: This is a kernel with Vybrid platform support fetched from <https://github.com/Timesys/linux-timesys>

As we mentioned before, all recipes provide the package name (for example, `linux-imx` or `linux-fslc`) by default, but all Linux recipes must also provide the `virtual/kernel` virtual package. The build system will resolve `virtual/kernel` to the most appropriate Linux recipe name, taking into account the requirements of the build, such as the machine it is building for.

And within those recipes, `linux-imx`, for example, has both 2.6.35.3 and 3.10.17 recipe versions.

In this recipe, we will show how to tell the Yocto build system which specific package and version to build.

How to do it...

To specify the exact package we want to build, the build system allows us to specify what provider and version to use.

How do we select which provider to use?

We can tell BitBake which recipe to use by using the `PREFERRED_PROVIDER` variable. To set a preferred provider for the `virtual/kernel` virtual package on our Wandboard machine, we would add the following to its machine configuration file:

```
PREFERRED_PROVIDER_virtual/kernel = "linux-imx"
```

How do we select which version to use?

Within a specific provider, we can also tell BitBake which version to use with the `PREFERRED_VERSION` variable. For example, to set a specific `linux-imx` version for all i.MX6-based machines, we would add the following to our `conf/local.conf` file:

```
PREFERRED_VERSION_linux-imx_mx6 = "3.10.17"
```

The `%` wildcard is accepted to match any character, as we see here:

```
PREFERRED_VERSION_linux-imx_mx6 = "3.10%"
```

It is, however, more common to see this type of configuration done in machine configuration files, in which case we would not use the `_mx6` append.

How do we select which version not to use?

We can use the `DEFAULT_PREFERENCE` variable set to `-1` to specify that a version is not to be used unless explicitly set by a `PREFERRED_VERSION` variable. This is commonly used in development versions of packages.

```
DEFAULT_PREFERENCE = "-1"
```

Adding supported packages

It is common to want to add new packages to an image that already has an available recipe in one of the included Yocto layers.

When the target image desired is very different from the supplied core images, it is recommended to define a new image rather than to customize an existing one.

This recipe will show how to customize an existing image by adding supported packages to it, but also to create a completely new image recipe if needed.

Getting ready

To discover whether a package we require is included in our configured layers, and what specific versions are supported, we can use `bitbake-layers` from our build directory as we saw previously:

```
$ bitbake-layers show-recipes | grep -A 1 htop
htop:
    meta-oe           1.0.3
```

Alternatively, we can also use BitBake as follows:

```
$ bitbake -s | grep htop
htop                         :1.0.3-r0
```

Or we can use the `find` Linux command in our sources directory:

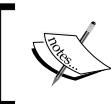
```
$ find . -type f -name "htop*.bb"
./meta-openembedded/meta-oe/recipes-support/htop/htop_1.0.3.bb
```

Once we know what packages we want to include in our final images, let's see how we can add them to the image.

How to do it...

While developing, we will use our project's `conf/local.conf` file to add customizations. To add packages to all images, we can use the following line of code:

```
IMAGE_INSTALL_append = " htop"
```



Note that there is a space after the first quote to separate the new package from the existing ones, as the `append` operator does not add a space.

We could also limit the addition to a specific image with:

```
IMAGE_INSTALL_append_pn-core-image-minimal = " htop"
```

Another way to easily customize is by making use of **features**. A feature is a logical grouping of packages. For example, we could create a new feature called `debug-utils`, which will add a whole set of debugging utilities. We could define our feature in a configuration file or class as follows:

```
FEATURE_PACKAGES_debug-utils = "strace perf"
```

We could then add this feature to our image by adding an `EXTRA_IMAGE_FEATURES` variable to our `conf/local.conf` file as follows:

```
EXTRA_IMAGE_FEATURES += "debug-utils"
```

If you were to add it to an image recipe, you would use the `IMAGE_FEATURES` variable instead.

Usually, features get added as a `packagegroup` recipe instead of being listed as packages individually. Let's show how to define a `packagegroup` recipe in the `recipes-core/packagegroups/packagegroup-debug-utils.bb` file:

```
SUMMARY = "Debug applications packagegroup"
LICENSE = "GPLv2"
LIC_FILES_CHKSUM = "file://${COREBASE}/LICENSE;md5=3f40d7994397109285e
c7b81fdeb3b58"

inherit packagegroup

RDEPENDS_${PN} = "\\
    strace \
    perf \
"
```

And you would then add it to the `FEATURE_PACKAGES` variable as follows:

```
FEATURE_PACKAGES_debug-utils = "packagegroup-debug-utils"
```

We can use `packagegroups` to create more complex examples. Refer to the *Yocto Project Development Manual* at <http://www.yoctoproject.org/docs/1.7.1/dev-manual/dev-manual.html> for details.

How it works...

The best approach to customize images is to create our own images using an existing image as template. We could use `core-image-minimal.bb`, which contains the following code:

```
SUMMARY = "A small image just capable of allowing a device to
boot."

IMAGE_INSTALL = "packagegroup-core-boot
${ROOTFS_PKGMANAGE_BOOTSTRAP} ${CORE_IMAGE_EXTRA_INSTALL}"

IMAGE_LINGUAS = " "

LICENSE = "MIT"

inherit core-image

IMAGE_ROOTFS_SIZE ?= "8192"
```

And extend it to your own version that allows for the customization of `IMAGE_FEATURES`, by adding the following `meta-custom/recipes-core/images/custom-image.bb` image file:

```
require recipes-core/images/core-image-minimal.bb
IMAGE_FEATURES += "ssh-server-dropbear package-management"
```

Of course, we can also define a new image from scratch using one of the available images as a template.

There's more...

A final way to customize images is by adding shell functions that get executed once the image has been created. You do this by adding the following to your image recipe or `conf/local.conf` file:

```
ROOTFS_POSTPROCESS_COMMAND += "function1;...;functionN"
```

You can use the path to the root filesystem in your command with the `IMAGE_ROOTFS` variable.

Classes would use the `IMAGE_POSTPROCESS_COMMAND` variable instead of `ROOTFS_POSTPROCESS_COMMAND`.

One example of usage can be found in the `debug-tweaks` feature in `image.bbclass`, when images are tweaked to allow passwordless root logins. This method is also commonly used to customize the root password of a target image.

Configuring packages

As we saw in the *Configuring the Linux kernel* recipe in *Chapter 2, The BSP Layer*, some packages, like the Linux kernel, provide a configuration menu and can be configured with the `menuconfig` BitBake command.

Another package worth mentioning with a configuration interface is BusyBox. We will show how to configure BusyBox, for example to add `pgrep`, a tool that looks up process's IDs by name. To do so follow the next steps:

1. Configure BusyBox:

```
$ bitbake -c menuconfig busybox
```

2. In **Process utilities** choose `pgrep`.

3. Compile BusyBox:

```
$ bitbake -C compile busybox
```

4. Copy the RPM package into the target:

```
$ bitbake -e busybox | grep ^WORKDIR=
$ scp ${WORKDIR}/deploy-rpms/cortexa9hf_vfp_neon/busybox-
1.22.1-r32.cortexa9hf_vfp_neon.rpm root@<target_ip>:/tmp
```

5. Install the RPM package on the target:

```
# rpm --force -U /tmp/busybox-1.22.1-
r32.cortexa9hf_vfp_neon.rpm
```

Note that we are forcing the update as the package version has not increased with the configuration change.

Adding new packages

We have seen how to customize our image so that we can add supported packages to it. When we can't find an existing recipe or we need to integrate some new software we have developed, we will need to create a new Yocto recipe.

Getting ready

There are some questions we need to ask ourselves before starting to write a new recipe:

- ▶ Where is the source code stored?
- ▶ Is it source-controlled or released as a tarball?
- ▶ What is the source code license?
- ▶ What build system is it using?

- ▶ Does it need configuration?
- ▶ Can we cross-compile it as is or does it need to be patched?
- ▶ What are the files that need to be deployed to the root filesystem, and where do they go?
- ▶ Are there any system changes that need to happen, such as new users or init scripts?
- ▶ Are there any dependencies that need to be installed into `sysroot` beforehand?

Once we know the answers to these questions, we are ready to start writing our recipe.

How to do it...

It is best to start from a blank template like the one that follows than to start from a similar recipe and modify it, as the result will be cleaner and contain only the strictly needed instructions.

A good starting base for a minimal recipe addition is:

```
SUMMARY = "The package description for the package management
system"

LICENSE = "The package's licenses typically from
meta/files/common-licenses/"
LIC_FILES_CHKSUM = "License checksum used to track open license
changes"
DEPENDS = "Package list of build time dependencies"

SRC_URI = "Local or remote file or repository to fetch"
SRC_URI[md5sum] = "md5 checksums for all remote fetched files (not
for repositories)"
SRC_URI[sha256sum] = "sha256 checksum for all remote fetched files
(not for repositories)"

S = "Location of the source in the working directory, by default
${WORKDIR}/${PN}-${PV}."

inherit <class needed for some functionality>

# Task overrides, like do_configure, do_compile and do_install, or
nothing.

# Package splitting (if needed).

# Machine selection variables (if needed).
```

How it works...

We will explain each one of the recipe sections in more detail in the following sections.

Package licensing

Every recipe needs to contain a `LICENSE` variable. The `LICENSE` variable allows you to specify multiple, alternative, and per-package type licenses, as seen in the following examples:

- ▶ For MIT or GPLv2 alternative licenses, we will use:

```
LICENSE = "GPL-2.0 | MIT"
```

- ▶ For both ISC and MIT licenses, we will use:

```
LICENSE = "ISC & MIT"
```

- ▶ For split packages, all of them GPLv2 except the documentation that is covered under the Creative Commons, we will use:

```
LICENSE_${PN} = "GPLv2"  
LICENSE_${PN}-dev = "GPLv2"  
LICENSE_${PN}-dbg = "GPLv2"  
LICENSE_${PN}-doc = "CC-BY-2.0"
```

Open source packages usually have the license included with the source code in `README`, `COPYING`, or `LICENSE` files, and even the source code header files.

For open source licenses, we also need to specify `LIC_FILES_CHECKSUM` for all licenses so that the build system can notify us when the licenses change. To add it, we locate the file or file portion that contains the license and provide its relative path from the directory containing the source and a MD5 checksum for it. For example:

```
LIC_FILES_CHKSUM = "file://${COREBASE}/meta/files/common-  
licenses/GPL-2.0;md5=801f80980d171dd6425610833a22dbe6"  
LIC_FILES_CHKSUM =  
"file://COPYING;md5=f7bdc0c63080175d1667091b864cb12c"  
LIC_FILES_CHKSUM =  
"file://usr/include/head.h;endline=7;md5=861ebad4adc7236f8d1905338  
abd7eb2"  
LIC_FILES_CHKSUM =  
"file://src/file.c;beginline=5;endline=13;md5=6c7486b21a8524b1879f  
a159578da31e"
```

Proprietary code should have the license set as `CLOSED`, and no `LIC_FILES_CHECKSUM` is needed for it.

Fetching package contents

The `SRC_URI` variable lists the files to fetch. The build system will use different fetchers depending on the file prefix. These can be:

- ▶ Local files included with the metadata (`file://`). If the local file is a patch, the `SRC_URI` variable can be extended with patch-specific arguments such as:
 - `striplevel`: The default patch strip level is 1 but it can be modified with this argument
 - `patchdir`: This specifies the directory location to apply the patch to, with the default being the source directory
 - `apply`: This argument controls whether to apply the patch or not, with the default being to apply it
- ▶ Files stored in remote servers (typically, `http(s)://`, `ftp://`, or `ssh://`).
- ▶ Files stored in remote repositories (typically, `git://`, `svn://`, `hg://`, or `bzr://`). These also need a `SRCREV` variable to specify the revision.

Files stored in remote servers (not local files or remote repositories) need to specify two checksums. If there are several files, they can be distinguished with a `name` argument; for example:

```
SRCREV = "04024dea2674861fcf13582a77b58130c67fccd8"  
SRC_URI = "git://repo.com/git/ \  
          file://fix.patch;name=patch \  
          http://example.org/archive.data;name=archive"  
SRC_URI[archive.md5sum] = "aaf32bde135cf3815aa3221726bad71e"  
SRC_URI[archive.sha256sum] =  
  "65be91591546ef6fdfec93a71979b2b108eee25edbc20c53190caafc9a92d4e7"
```

The source directory folder, `s`, specifies the location of the source files. The repository will be checked out here, or the tarball decompressed in this location. If the tarball decompresses in the standard `$(PN)-$(PV)` location, it can be omitted as it is the default. For repositories, it needs to always be specified; for example:

```
S = "${WORKDIR}/git"
```

Specifying task overrides

All recipes inherit the `base.bbclass` class, which defines the following tasks:

- ▶ `do_fetch`: This method fetches the source code, selecting the fetcher using the `SRC_URI` variable.
- ▶ `do_unpack`: This method unpacks the code in the working directory to a location specified by the `s` variable.
- ▶ `do_configure`: This method configures the source code if needed. It does nothing by default.

- ▶ `do_compile`: This method compiles the source and runs the GNU make target by default.
- ▶ `do_install`: This method copies the results of the build from the `build` directory B to the destination directory D. It does nothing by default.
- ▶ `do_package`: This method splits the deliverables into several packages. It does nothing by default.

Usually, only the configuration, compilation, and installation tasks are overridden, and this is mostly done implicitly by inheriting a class like `autotools`.

For a custom recipe that does not use a build system, you need to provide the required instructions for configuration (if any), compilation, and installation in their corresponding `do_configure`, `do_compile`, and `do_install` overrides. As an example of this type of recipe, `meta-custom/recipes-example/helloworld/helloworld_1.0.bb`, may be seen here:

```
DESCRIPTION = "Simple helloworld application"
SECTION = "examples"
LICENSE = "MIT"
LIC_FILES_CHKSUM =
    "file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4
f302"

SRC_URI = "file://helloworld.c"

S = "${WORKDIR}"

do_compile() {
    ${CC} helloworld.c -o helloworld
}

do_install() {
    install -d ${D}${bindir}
    install -m 0755 helloworld ${D}${bindir}
}
```

With the `meta-custom/recipes-example/helloworld/helloworld_1.0/helloworld.c` source file being the following:

```
#include <stdio.h>

int main(void)
{
    return printf("Hello World");
}
```

We will see example recipes that use the most common build systems in the next chapter.

Configuring packages

The Yocto build system provides the PACKAGECONFIG variable to help in the configuration of packages by defining a number of features. Your recipe defines the individual features as follows:

```
PACKAGECONFIG ??= "feature"
PACKAGECONFIG[feature] = "--with-feature,--without-feature,build-
deps-feature,rt-deps-feature"
```

The PACKAGECONFIG variable contains a space-separated list of feature names, and it can be extended or overridden in bbappend files; have a look at the following example:

```
PACKAGECONFIG_append = " feature1 feature2"
```

To extend or override it from a distribution or local configuration file, you would use the following syntax:

```
PACKAGECONFIG_pn-<package_name> = "feature1 feature2"
PACKAGECONFIG_append_pn-<package_name> = " feature1 feature2"
```

Following that, we characterize each feature with four ordered arguments:

- ▶ Extra configuration arguments (for EXTRA_OECONF) when the feature is enabled
- ▶ Extra configuration arguments (for EXTRA_OECONF) when the feature is disabled
- ▶ Extra build dependencies (for DEPENDS) when the feature is enabled
- ▶ Extra runtime dependencies (for RDEPENDS) when the feature is enabled

The four arguments are optional, but the ordering needs to be maintained by leaving the surrounding commas.

For example, the wpa-suplicant recipe defines two features, gnutls and openssl, but only enables gnutls by default, as seen here:

```
PACKAGECONFIG ??= "gnutls"
PACKAGECONFIG[gnutls] = ",,gnutls"
PACKAGECONFIG[openssl] = ",,openssl"
```

Splitting into several packages

It is common to separate the recipe contents into different packages that serve different needs. Typical examples are to include documentation in a doc package, and header and/or libraries in a dev package. We can do this using the FILES variable as follows:

```
FILES_${PN} += "List of files to include in the main package"
FILES_${PN}-dbg += "Optional list of files to include in the debug
package"
FILES_${PN}-dev += "Optional list of files to include in the
development package"
FILES_${PN}-doc += "Optional list of files to include in the
documentation package"
```

Setting machine-specific variables

Each recipe has a `PACKAGE_ARCH` variable that categorizes the recipe into a package feed, as we saw in the *Exploring an image's contents* recipe. Most of the times, they are automatically sorted out by the Yocto build system. For example, if the recipe is a kernel, a kernel module recipe, or an image recipe, or even if it is cross-compiling or building native applications, the Yocto build system will set the package architecture accordingly.

BitBake will also look at the `SRC_URI` machine overrides and adjust the package architecture, and if your recipe is using the `allarch` class, it will set the package architecture to `all`.

So when working on a recipe that only applies to a machine or machine family, or that contains changes that are specific to a machine or machine family, we need to check whether the package is categorized in the appropriate package feed, and if not, specify the package architecture explicitly in the recipe itself by using the following line of code:

```
PACKAGE_ARCH = "${MACHINE_ARCH}"
```

Also, when a recipe is only to be parsed for specific machine types, we specify it with the `COMPATIBLE_MACHINE` variable. For example, to make it compatible only with the `mxs`, `mx5` and `mx6` SoC families, we would use the following:

```
COMPATIBLE_MACHINE = "(mxs|mx5|mx6)"
```

Adding data, scripts, or configuration files

All recipes inherit the base class with the default set of tasks to run. After inheriting the base class, a recipe knows how to do things like fetching and compiling.

As most recipes are meant to install some sort of executable, the base class knows how to build it. But sometimes all we want is to install data, scripts, or configuration files into the filesystem.

If the data or configuration is related to an application, the most logical thing to do is to package it together with the application's recipe itself, and if we think it is better to be installed separately, we could even split it into its own package.

But some other times, the data or configuration is unrelated to an application, maybe it applies to the whole system or we just want to provide a separate recipe for it. Optionally, we could even want to install some Perl or Python scripts that don't need to be compiled.

How to do it...

In those cases, our recipe should inherit the `allarch` class that is inherited by recipes that do not produce architecture-specific output.

An example of this type of recipe, `meta-custom/recipes-example/example-data/example-data_1.0.bb`, may be seen here:

```
DESCRIPTION = "Example of data or configuration recipe"
SECTION = "examples"

LICENSE = "GPLv2"
LIC_FILES_CHKSUM = "file://${COREBASE}/meta/files/common-licenses/GPL-2.0;md5=801f80980d171dd6425610833a22dbe6"

SRCREV = "${AUTOREV}"
SRC_URI = "git://github.com/yoctocookbook/examples.git \
           file://example.data"

S = "${WORKDIR}/git"

inherit allarch

do_compile() {
}

do_install() {
    install -d ${D}${sysconfdir}
    install -d ${D}${sbindir}
    install -m 0755 ${WORKDIR}/example.data ${D}/${sysconfdir}/
    install -m 0755 ${S}/python-scripts/* ${D}/${sbindir}
}
```

It assumes that the fictitious `examples.git` repository contains a `python-scripts` folder, which we want to include in our root filesystem.

A working recipe example can be found in the source that accompanies the book.

Managing users and groups

It is also common to need to add or modify users and groups to our filesystem. This recipe explains how it is done.

Getting ready

The user information is stored in the `/etc/passwd` file, a text file that is used as a database for the system user's information. The `passwd` file is human-readable.

Each line on it corresponds to one user in the system, and it has the following format:

```
<username>:<password>:<uid>:<gid>:<comment>:<home  
directory>:<login shell>
```

Let's see each of the parameters of this format:

- ▶ **username:** A unique string that identifies the user at login
- ▶ **uid:** User ID, a number that Linux uses to identify the user
- ▶ **gid:** Group ID, a number that Linux uses to identify the user's primary group
- ▶ **comment:** Comma-separated values that describe the account, typically the user's contact details
- ▶ **home directory:** Path to the user's home directory
- ▶ **login shell:** Shell that is started for interactive logins

The default `passwd` file is stored with the `base-passwd` package and looks as follows:

```
root::0:0:root:/root:/bin/sh  
daemon:*:1:1:daemon:/usr/sbin:/bin/sh  
bin:*:2:2:bin:/bin:/bin/sh  
sys:*:3:3:sys:/dev:/bin/sh  
sync:*:4:65534:sync:/bin:/bin/sync  
games:*:5:60:games:/usr/games:/bin/sh  
man:*:6:12:man:/var/cache/man:/bin/sh  
lp:*:7:7:lp:/var/spool/lpd:/bin/sh  
mail:*:8:8:mail:/var/mail:/bin/sh  
news:*:9:9:news:/var/spool/news:/bin/sh  
uucp:*:10:10:uucp:/var/spool/uucp:/bin/sh  
proxy:*:13:13:proxy:/bin:/bin/sh  
www-data:*:33:33:www-data:/var/www:/bin/sh  
backup:*:34:34:backup:/var/backups:/bin/sh  
list:*:38:38:Mailing List Manager:/var/list:/bin/sh  
irc:*:39:39:ircd:/var/run/ircd:/bin/sh  
gnats:*:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/bin/  
sh  
nobody:*:65534:65534:nobody:/nonexistent:/bin/sh
```

All accounts have disabled direct logins, indicated by an asterisk on the password field, except for root, which has no password. This is because, by default, the image is built with the `debug-tweaks` feature that enables passwordless login for the root user, among other things. If the root password was enabled, we would see the encrypted root password.



Do not forget to remove the `debug-tweaks` feature from production images.

There is a corresponding `/etc/group` file that is installed at the same time with the information for the system groups.

The `core-image-minimal` image does not include shadow password protection, but other images, such as `core-image-full-cmdline`, do. When enabled, all password fields contain an x, and the encrypted passwords are kept on a `/etc/shadow` file, which is only accessible to the super user.

Any user that is needed by the system but not included in the list we saw earlier needs to be created.

How to do it...

The standard way for a recipe to add or modify system users or groups is to use the `useradd` class, which uses the following variables:

- ▶ `USERADD_PACKAGES`: This variable specifies the individual packages in the recipe that require users or groups to be added. For the main package, you would use the following:
`USERADD_PACKAGES = "${PN}"`
- ▶ `USERADD_PARAM`: This variable corresponds to the arguments passed to the Linux `useradd` command, to add new users to the system.
- ▶ `GROUPADD_PARAM`: This variable corresponds to the arguments passed to the Linux `groupadd` command, to add new groups to the system.
- ▶ `GROUPMEMS_PARAM`: This variable corresponds to the arguments passed to the Linux `groupmems` command, which administers members of the user's primary group.

An example snippet of a recipe using the `useradd` class follows:

```
inherit useradd

PASSWORD ?= "miDBHFo2hJSAA"
USERADD_PACKAGES = "${PN}"
USERADD_PARAM_${PN} = "--system --create-home \
                     --groups tty \
                     --password ${PASSWORD} \
                     --user-group ${PN}"
```

The password can be generated on your host using the `mkpasswd` Linux command-line utility, installed with the `whois` Ubuntu package.

There's more...

When generating users and groups using the `useradd` class, the `uid` and `gid` values are assigned dynamically during package installation. If this is not desired, there is a way to assign system-wide static `uid` and `gid` values by providing your own `passwd` and group files.

To do this, you need to define the `USERADDEXTRACTION` variable in your `conf/local.conf` file as follows:

```
USERADDEXTRACTION = "useradd-staticids"
```

The build system will then search the `BBPATH` variable for `files/passwd` and `files/group` files to obtain the `uid` and `gid` values. The files have the standard `passwd` layout as defined previously, with the password field ignored.

The default filenames can be overridden by using the `USERADD_UID_TABLES` and `USERADD_GID_TABLES` variables.

You also need to define the following:

```
USERADD_ERROR_DYNAMIC = "1"
```

This is done so that the build system produces an error if the required `uid` and `gid` values are not found in the provided files.



Note that if you use the `useradd` class in a project that is already built, you will need to remove the `tmp` directory and rebuild from the `sstate-cache` directory or you will get build errors.

There is also a way to add user and group information that is not tied to a specific recipe but to an image – by using the `extrausers` class. It is configured by the `EXTRA_USERS_PARAMS` variable in an image recipe and used as follows:

```
inherit extrausers

EXTRA_USERS_PARAMS = "\\\n    useradd -P password root; \\\n    "
```

This sets the root password to `password`.

Using the sysvinit initialization manager

The initialization manager is an important part of the root filesystem. It is the first thing the kernel executes, and it has the responsibility to start the rest of the system.

This recipe will introduce the sysvinit initialization manager.

Getting ready

This is the default initialization manager in Yocto and it has been used in Linux since the operating system's origin. The kernel is passed an `init` command-line argument, typically `/sbin/init`, which is then launched. This `init` process has PID 1 and is the parent of all processes. The `init` process can either be implemented by BusyBox or be an independent program installed with the `sysvinit` package. Both of them work in the same way, based on the concept of **runlevel**, a machine state that defines which processes to run.

The `init` process will read an `inittab` file and look for a default runlevel. The default `inittab` file is installed with the `sysvinit-inittab` package and is as follows:

```
# /etc/inittab: init(8) configuration.
# $Id: inittab,v 1.91 2002/01/25 13:35:21 miquels Exp $

# The default runlevel.
id:5:initdefault:

# Boot-time system configuration/initialization script.
# This is run first except when booting in emergency (-b) mode.
si::sysinit:/etc/init.d/rcS

# What to do in single-user mode.
~~:S:wait:/sbin/sulogin

# /etc/init.d executes the S and K scripts upon change
# of runlevel.
#
# Runlevel 0 is halt.
# Runlevel 1 is single-user.
# Runlevels 2-5 are multi-user.
# Runlevel 6 is reboot.

10:0:wait:/etc/init.d/rc 0
11:1:wait:/etc/init.d/rc 1
12:2:wait:/etc/init.d/rc 2
13:3:wait:/etc/init.d/rc 3
```

```
14:4:wait:/etc/init.d/rc 4
15:5:wait:/etc/init.d/rc 5
16:6:wait:/etc/init.d/rc 6
# Normally not reached, but fallthrough in case of emergency.
z6:6:respawn:/sbin/sulogin
```

Then, init runs all scripts starting with `s` in the `/etc/rcS.d` directory, followed by all the scripts starting with `s` in the `/etc/rcN.d` directory, where `N` is the runlevel value.

So the init process just performs the initialization and forgets about the processes. If something goes wrong and the processes are killed, no one will care. The system watchdog will reboot the system if it becomes unresponsive, but applications built with more than one process usually need some type of process monitor that can react to the health of the system, but sysvinit does not offer these types of mechanisms.

However, sysvinit is a well-understood and reliable initialization manager and the recommendation is to keep it unless you need some extra feature.

How to do it...

When using sysvinit as the initialization manager, Yocto offers the `update-rc.d` class as a helper to install initialization scripts so that they are started and stopped when needed.

When using this class, you need to specify the `INITSCRIPT_NAME` variable with the name of the script to install and `INITSCRIPT_PARAMS` with the options to pass to the `update-rc.d` utility. You can optionally use the `INITSCRIPT_PACKAGES` variable to list the packages to contain the initialization scripts. By default, this contains the main package only, and if multiple packages are provided, the `INITSCRIPT_NAME` and `INITSCRIPT_PARAMS` need to be specified for each using overrides. An example snippet follows:

```
INITSCRIPT_PACKAGES = "${PN}-httpd ${PN}-ftpd"
INITSCRIPT_NAME_${PN}-httpd = "httpd.sh"
INITSCRIPT_NAME_${PN}-ftpd = "ftpd.sh"
INITSCRIPT_PARAMS_${PN}-httpd = "defaults"
INITSCRIPT_PARAMS_${PN}-ftpd = "start 99 5 2 . stop 20 0 1 6 ."
```

When an initialization script is not tied to a particular recipe, we can add a specific recipe for it. For example, the following recipe will run a `mount.sh` script in the `recipes-example/sysvinit-mount`/`sysvinit-mount_1.0.bb` file.

```
DESCRIPTION = "Initscripts for mounting filesystems"
LICENSE = "MIT"

LIC_FILES_CHKSUM =
    "file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4
f302"
```

```
SRC_URI = "file://mount.sh"

INITSCRIPT_NAME = "mount.sh"
INITSCRIPT_PARAMS = "start 09 S ."

inherit update-rc.d

S = "${WORKDIR}"

do_install () {
    install -d ${D}${sysconfdir}/init.d/
    install -c -m 755 ${WORKDIR}/${INITSCRIPT_NAME}
    ${D}${sysconfdir}/init.d/${INITSCRIPT_NAME}
}
```

Using the systemd initialization manager

As an alternative to `sysvinit`, you can configure your project to use `systemd` as an initialization manager, although `systemd` packs many more features.

Getting ready

The `systemd` initialization manager is replacing `sysvinit` and other initialization managers in most Linux distributions. It is based on the concepts of units, an abstraction of all elements that are relevant for system startup and maintenance, and targets, which group units and can be viewed as a runlevel equivalent. Some of the units `systemd` defines are:

- ▶ Services
- ▶ Sockets
- ▶ Devices
- ▶ Mount points
- ▶ Snapshots
- ▶ Timers
- ▶ Paths

The default targets and their runlevel equivalents are defined in the following table:

Sysvinit	Runlevel	Systemd target	Notes
0	runlevel0. target	poweroff.target	Halt the system.
1, s, single	runlevel1. target	rescue.target	Single user mode.

Sysvinit	Runlevel	Systemd target	Notes
2, 4	runlevel2. target, runlevel4. target	multi-user. target	User-defined/site-specific runlevels. By default, identical to 3.
3	runlevel3. target	multi-user. target	Multiuser, non-graphical. Users can usually log in via multiple consoles or via the network.
5	runlevel5. target	graphical. target	Multiuser, graphical. Usually has all the services of runlevel 3 plus a graphical login.
6	runlevel6. target	reboot.target	Reboot the system.

The `systemd` initialization manager is designed to be compatible with `sysvinit`, including using `sysvinit` init scripts.

Some of the features of `systemd` are:

- ▶ Parallelization capabilities that allow for faster boot times
- ▶ Service initialization via sockets and D-Bus so that services are only started when needed
- ▶ Process monitoring that allows for process failure recovery
- ▶ System state snapshots and restoration
- ▶ Mount point management
- ▶ Transactional-dependency-based unit control, where units establish dependencies between them

How to do it...

To configure your system to use `systemd`, you need to add the `systemd` distribution feature to your project by adding the following to your distribution's configuration file, under `sources/poky/meta-yocto/conf/distro/poky.conf` for the default `poky` distribution, or locally on your project's `conf/local.conf` file:

```
DISTRO_FEATURES_append = " systemd"
```



Note the space required after the starting quote.

```
VIRTUAL-RUNTIME_init_manager = "systemd"
```

This configuration example allows you to define a main image with `systemd` and a rescue image with `sysvinit`, providing it does not use the `VIRTUAL-RUNTIME_init_manager` variable. Hence, the rescue image cannot use the `packagegroup-core-boot` or `packagegroup-core-full-cmdline` recipes. As an example, the recipe where the image size has been reduced, which we will introduce in the *Reducing the root filesystem image size* recipe in this chapter, could be used as the basis for a rescue image.

To remove `sysvinit` completely from your system, you would do the following:

```
DISTRO_FEATURES_BACKFILL_CONSIDERED = "sysvinit"  
VIRTUAL-RUNTIME_initscripts = ""
```

Feature backfilling is the automatic extension of machine and distribution features to keep backwards compatibility. The `sysvinit` distribution feature is automatically filled in, so to remove it, we need to blacklist it by adding it to the `DISTRO_FEATURES_BACKFILL_CONSIDERED` variable as shown earlier.



Note that if you are using an existing project and you change the `DISTRO_FEATURES` variable as explained earlier, you will need to remove the `tmp` directory and build with `sstate-cache` or the build will fail.

There's more...

Not only does the root filesystem needs to be configured, but the Linux kernel also needs to be specifically configured with all the features required by `systemd`. There is an extensive list of kernel configuration variables in the `systemd` source README file. As an example, to extend the minimal kernel configuration that we will introduce in the *Reducing the Linux kernel image size* recipe later on this chapter, for the Wandboard to support `systemd`, we would need to add the following configuration changes in the `arch/arm/configs/wandboard-quad_minimal_defconfig` file:

```
+CONFIG_FHANDLE=y  
+CONFIG_CGROUPS=y  
+CONFIG_SECCOMP=y  
+CONFIG_NET=y  
+CONFIG_UNIX=y  
+CONFIG_INET=y  
+CONFIG_AUTOFS4_FS=y  
+CONFIG_TMPFS=y  
+CONFIG_TMPFS_POSIX_ACL=y  
+CONFIG_SCHEDSTATS=y
```

The default kernel configuration provided for the Wandboard will launch a `core-image-minimal` image of `systemd` just fine.

Installing systemd unit files

Yocto offers the `systemd` class as a helper to install unit files. By default, unit files are installed on the `$(systemd_unitdir)/system` path on the destination directory.

When using this class, you need to specify the `SYSTEMD_SERVICE_${PN}` variable with the name of the unit file to install. You can optionally use the `SYSTEMD_PACKAGES` variable to list the packages to contain the unit files. By default, this is the main package only, and if multiple packages are provided, the `SYSTEMD_SERVICE` variable needs to be specified using overrides.

Services are configured to launch at boot by default, but this can be changed with the `SYSTEMD_AUTO_ENABLE` variable.

An example snippet follows:

```
SYSTEMD_PACKAGES = "${PN}-syslog"
SYSTEMD_SERVICE_${PN}-syslog = "busybox-syslog.service"
SYSTEMD_AUTO_ENABLE = "disabled"
```

Installing package-installation scripts

The supported package formats, RPM, ipk, and deb, support the addition of installation scripts that can be run at different times during a package installation process. In this recipe, we will see how to install them.

Getting ready

There are different types of installation scripts:

- ▶ **Preinstallation scripts** (`pkg_preinst`): These are called before the package is unpacked
- ▶ **Postinstallation scripts** (`pkg_postinst`): These are called after the package is unpacked, and dependencies will be configured
- ▶ **Preremoval scripts** (`pkg_prerm`): These are called with installed or at least partially installed packages
- ▶ **Postremoval scripts** (`pkg_postrm`): These are called after the package's files have been removed or replaced

How to do it...

An example snippet of the installation of a preinstallation script in a recipe is as follows:

```
pkg_preinst_${PN} () {
    # Shell commands
}
```

All installation scripts work in the same way, with the exception that the postinstallation scripts may be run either on the host at root filesystem image creation time, or on the target (for those actions that cannot be performed on the host), or when a package is directly installed on the target. Have a look at the following code:

```
pkg_postinst_${PN} () {
    if [ x"${D}" = "x" ]; then
        # Commands to execute on device
    else
        # Commands to execute on host
    fi
}
```

If the postinstallation script succeeds, the package is marked as installed. If the script fails, the package is marked as unpacked and the script is executed when the image boots again.

How it works...

Once the recipe defines an installation script, the class for the specific package type will install it while following the packaging rules of the specific format.

For postinstallation scripts, when running on the host, `D` is set to the destination directory, so the comparison test will fail. But `D` will be empty when running on the target.



It is recommended to perform postinstallation scripts on the host if possible, as we need to take into account that some root filesystems will be read only and hence it would not be possible to perform some operations on the target.

Reducing the Linux kernel image size

Before or in parallel with the root filesystem customization, embedded projects usually require an image size optimization that will reduce the boot time and memory usage.

Smaller images mean less storage space, less transmission time, and less programming time, which saves money both in manufacturing and field updates.

By default, the compressed Linux kernel image (`zImage`) for the wandboard-quad is around 5.2 MB. This recipe will show how we can reduce that.

How to do it...

An example of a minimal kernel configuration for a Wandboard that is able to boot from a microSD card root filesystem is the `arch/arm/configs/wandboard-quad_minimal_defconfig` file that follows:

```
CONFIG_KERNEL_XZ=y
CONFIG_NO_HZ=y
CONFIG_HIGH_RES_TIMERS=y
CONFIG_BLK_DEV_INITRD=y
CONFIG_CC_OPTIMIZE_FOR_SIZE=y
CONFIG_EMBEDDED=y
CONFIG_SLOB=y
CONFIG_ARCH_MXC=y
CONFIG_SOC_IMX6Q=y
CONFIG_SOC_IMX6SL=y
CONFIG_SMP=y
CONFIG_VMSPLIT_2G=y
CONFIG_AEABI=y
CONFIG_CPU_FREQ=y
CONFIG_ARM_IMX6_CPUFREQ=y
CONFIG_CPU_IDLE=y
CONFIG_VFP=y
CONFIG_NEON=y
CONFIG_DEV TMPFS=y
CONFIG_DEV TMPFS_MOUNT=y
CONFIG PROC_DEVICETREE=y
CONFIG SERIAL_IMX=y
CONFIG SERIAL_IMX_CONSOLE=y
CONFIG_REGULATOR=y
CONFIG_REGULATOR_ANATOP=y
CONFIG_MMCA=y
CONFIG_MMCA_SDHCI=y
CONFIG_MMCA_SDHCI_PLTFM=y
CONFIG_MMCA_SDHCI_ESDHC_IMX=y
CONFIG_DMADEVICES=y
CONFIG_IMX_SDMA=y
CONFIG_EXT3_FS=y
```

This configuration builds an 886 K compressed Linux kernel image (`zImage`).

How it works...

Apart from hardware design considerations (such as running the Linux kernel from a NOR flash and **execute in place (XIP)** to avoid loading the image to memory), the first step in kernel size optimization is to review the kernel configuration and remove all superfluous features.

To analyze the sizes of kernel blocks, we may use:

```
$ size vmlinux */built-in.o
text      data      bss      dec      hex filename
8746205  356560  394484  9497249  90eaal vmlinux
117253    2418     1224   120895  1d83f block/built-in.o
243859    11158      20   255037  3e43d crypto/built-in.o
2541356   163465  34404  2739225  29cc19 drivers/built-in.o
1956        0       0    1956    7a4 firmware/built-in.o
1728762   18672  10544  1757978  1ad31a fs/built-in.o
20361    14701     100   35162   895a init/built-in.o
29628      760      8   30396   76bc ipc/built-in.o
576593   20644  285052  882289  d7671 kernel/built-in.o
106256   24847     2344  133447  20947 lib/built-in.o
291768   14901    3736  310405  4bc85 mm/built-in.o
1722683   39947  50928  1813558  1bac36 net/built-in.o
34638      848     316   35802   8bda security/built-in.o
276979   19748    1332  298059  48c4b sound/built-in.o
138        0       0    138     8a usr/built-in.o
```

Here, `vmlinux` is the Linux kernel ELF image, which can be found in the Linux build directory.

Some of the usual things to exclude are:

- ▶ Remove IPv6 (`CONFIG_IPV6`) and other superfluous networking features
- ▶ Remove block devices (`CONFIG_BLOCK`) if not needed
- ▶ Remove cryptographic features (`CONFIG_CRYPTO`) if unused
- ▶ Review the supported filesystem types and remove the unneeded ones, such as flash filesystems on flashless devices
- ▶ Avoid modules and remove the module support (`CONFIG_MODULES`) from the kernel if possible

A good strategy is to start with a minimal kernel and add the essential stuff until you get a working system. Start with the `allnoconfig` GNU make target and review the configuration items under `CONFIG_EXPERT` and `CONFIG_EMBEDDED` as they are not included in the `allnoconfig` setting.

Some configuration changes that might not be obvious but reduce the image size considerably without feature removal are listed here:

- ▶ Change the default compression method from **Lempel–Ziv–Oberhumer (LZO)** to XZ (`CONFIG_KERNEL_XZ`). The decompression speed will be a bit lower though.
- ▶ Change the allocator from SLUB to **Simple List Of Blocks (SLOB)** (`CONFIG_SLOB`) for small embedded systems with little memory.
- ▶ Use no high memory (`CONFIG_HIGHMEM`) unless you have 4 GB or more memory.

You may also want to have a different configuration for production and development systems, so you may remove the following from your production images:

- ▶ `printk` support (`CONFIG_PRINTK`)
- ▶ tracing support (`CONFIG_FTRACE`)

In the compilation side of things, optimize for size using `CONFIG_CC_OPTIMIZE_FOR_SIZE`.

Once the basics are covered, we would need to analyze the kernel functions to identify further reduction areas. You can print a sorted list of kernel symbols with the following:

```
$ nm --size-sort --print-size -r vmlinux | head  
808bde04 00040000 B __log_buf  
8060f1c0 00004f15 r kernel_config_data  
80454190 000041f0 T hidinput_connect  
80642510 00003d40 r drm_dmt_modes  
8065cbbc 00003414 R v4l2_dv_timings_presets  
800fbe44 000032c0 T __blockdev_direct_IO  
80646290 00003100 r edid_cea_modes  
80835970 00003058 t imx6q_clocks_init  
8016458c 00002e74 t ext4_fill_super  
8056a814 00002aa4 T hci_event_packet
```

You would then need to look into the kernel source to find optimizations.

The actual space used by the uncompressed kernel in memory can be obtained from a running Wandboard kernel log as follows:

```
$ dmesg | grep -A 3 "text"
  .text : 0x80008000 - 0x80a20538      (10338 kB)
  .init : 0x80a21000 - 0x80aae240      ( 565 kB)
  .data : 0x80ab0000 - 0x80b13644      ( 398 kB)
  .bss  : 0x80b13644 - 0x80b973fc      ( 528 kB)
```

From here, the `.text` section contains code and constant data, the `.data` section contains the initialization data for variables, and the `.bss` sections contains all uninitialized data. The `.init` section contains global variables used during Linux initialization only, which are freed afterwards as can be seen from the following Linux kernel boot message:

```
Freeing unused kernel memory: 564K (80a21000 - 80aae000)
```

There are ongoing efforts to reduce the size of the Linux kernel, so it is expected that newer kernel versions will be smaller and will allow for better customization for use in embedded systems.

Reducing the root filesystem image size

By default, the `core-image-minimal` size for the wandboard-quad unpacked tarball is around 45 MB, and `core-image-sato` is around 150 MB. This recipe will explore methods to reduce their size.

How to do it...

An example of a small image, `core-image-small`, that does not include the `packagegroup-core-boot` recipe and can be used as the base for a root filesystem image with reduced size, `recipes-core/images/core-image-small.bb`, is shown next:

```
DESCRIPTION = "Minimal console image."

IMAGE_INSTALL= \
    base-files \
    base-passwd \
    busybox \
    sysvinit \
    initscripts \
    ${ROOTFS_PKGMANAGE_BOOTSTRAP} \
    ${CORE_IMAGE_EXTRA_INSTALL} \
"
```

```
IMAGE_LINGUAS = " "
LICENSE = "MIT"
inherit core-image
IMAGE_ROOTFS_SIZE ?= "8192"
```

This recipe produces an image of about 6.4 MB. You can go even smaller if you use the poky-tiny distribution by adding the following to your `conf/local.conf` file:

```
DISTRO = "poky-tiny"
```

The poky-tiny distribution makes a series of size optimizations that may restrict the set of packages you can include in your image. To successfully build this image, you have to skip one of the sanity checks that the Yocto build system performs, by adding the following:

```
INSANE_SKIP_glibc-locale = "installed-vs-shipped"
```

With poky-tiny, the size of the image is further reduced to around 4 MB.

There are further reductions that can be done to the image; for example, we could replace `sysvinit` with `tiny-init`, but that is left as an exercise for the reader.

Images with reduced sizes are also used alongside production images for tasks such as rescue systems and manufacturing test processes. They are also ideal to be built as `initramfs` images; that is, images that the Linux kernel mounts from memory, and can even be bundled into a single Linux kernel image binary.

How it works...

Start with an appropriate image like `core-image-minimal` and analyze the dependencies as shown in the *Debugging the build system* recipe in *Chapter 1, The Build System*, and decide which of them are not needed. You could also use the file sizes listed in the image's build history, as seen in the *Using build history* recipe, also in *Chapter 1, The Build System*, to detect the biggest files in the filesystem and review them. To sort the file sizes, which appear in the fourth column of the `files-in-image.txt` file, in reverse order, we could execute:

```
$ sort -r -g -k 4,4 files-in-image.txt -o sorted-files-in-image.txt
sorted-files-in-image.txt:
-rwxr-xr-x root      root      1238640 ./lib/libc-2.19.so
-rwxr-xr-x root      root      613804 ./sbin/ldconfig
-rwxr-xr-x root      root      539860 ./bin/busybox.nosuid
-rwxr-xr-x root      root      427556 ./lib/libm-2.19.so
```

```
-rwxr-xr-x root      root      130304 ./lib/ld-2.19.so
-rwxr-xr-x root      root      88548 ./lib/libpthread-2.19.so
-rwxr-xr-x root      root      71572 ./lib/libnsl-2.19.so
-rwxr-xr-x root      root      71488 ./lib/libresolv-2.19.so
-rwsr-xr-x root      root      51944 ./bin/busybox.suid
-rwxr-xr-x root      root      42668 ./lib/libnss_files-
2.19.so
-rwxr-xr-x root      root      30536 ./lib/libnss_compat-
2.19.so
-rwxr-xr-x root      root      30244 ./lib/libcrypt-2.19.so
-rwxr-xr-x root      root      28664 ./sbin/init.sysvinit
-rwxr-xr-x root      root      26624 ./lib/librt-2.19.so
```

From this, we observe that `glibc` is the biggest contributor to the filesystem size. Some other places where some space on a console-only system can be saved are:

- ▶ Use the IPK package manager, as it is the lightest, or better yet, remove the package-management feature from your production root filesystem altogether.
- ▶ Use BusyBox's `mdev` device manager instead of `udev` by specifying it in your `conf/local.conf` file as follows:

```
VIRTUAL-RUNTIME_dev_manager = "mdev"
```

Note that this will only work with core images that include `packagegroup-core-boot`.

- ▶ If we are running the root filesystem on a block device, use `ext2` instead of `ext3` or `ext4` without the journal.
- ▶ Configure BusyBox with only the essential applets by providing your own configuration file in `bbappend`.
- ▶ Review the `glibc` configuration, which can be changed via the `DISTRO_FEATURES_LIBC` distribution configuration variable. An example of its usage can be found in the `poky-tiny` distribution, which is included in the `poky` source. The `poky-tiny` distribution can be used as a template for the distribution customization of small systems.
- ▶ Consider switching to a lighter C library than the default `glibc`. For a while, `uclibc` was being used as an alternative, but the library seems to be unmaintained for the last couple of years, and the `core-image-minimal` image for the Wandboard does not currently build using it.

Recently, there has been some activity with **musl** (<http://www.musl-libc.org/>), a new MIT-licensed C library. To enable it, you would add the following to your `conf/local.conf` file:

 `TCLIBC = "musl"`

And you would need to add the `meta-musl` layer (<https://github.com/kraj/meta-musl>) to your `conf/bblayers.conf` file.

It currently builds `core-image-minimal` for QEMU targets, but there is still work to be done to use it on real hardware like the Wandboard.

- ▶ Compile your applications with `-Os` to optimize for size.

Releasing software

When releasing a product based on the Yocto project, we have to consider that we are building on top of a multitude of different open source projects, each with different licensing requirements.

At the minimum, your embedded product will contain a bootloader (probably U-Boot), the Linux kernel, and a root filesystem with one or more applications. Both U-Boot and the Linux kernel are licensed under the **General Public License version 2 (GPLv2)**. And the root filesystem could contain a variety of programs with different licenses.

All open source licenses allow you to sell a commercial product with a mixture of proprietary and open licenses as long as they are independent and the product complies with all the open source licenses. We will discuss open source and proprietary cohabiting in the *Working with open source and proprietary code* recipe later on.

It is important to understand all the licensing implications before releasing your product to the public. The Yocto project provides tools to make handling licensing requirements an easier job.

Getting ready

We first need to specify what requirements we need to comply with to distribute a product built with the Yocto project. For the most restrictive open source licenses, this usually means:

- ▶ Source code distribution, including modifications
- ▶ License texts distributions
- ▶ Distribution of the tools used to build and run the software

How to do it...

We can use the archiver class to provide the deliverables that need to be distributed to comply with the licenses. We can configure it to:

- ▶ Provide the original unpatched source as tarballs
- ▶ Provide the patches to apply to the original source
- ▶ Provide the recipes used to build the source
- ▶ Provide the license text that must sometimes accompany the binary (according to some licenses)

To use the archiver class as specified earlier, we add the following to our conf/local.conf file:

```
INHERIT += "archiver"
ARCHIVER_MODE[src] = "original"
ARCHIVER_MODE[diff] = "1"
ARCHIVER_MODE[recipe] = "1"
COPY_LIC_MANIFEST = "1"
COPY_LIC_DIRS = "1"
```

The sources will be provided in the tmp/deploy/sources directory under a license subdirectory hierarchy.

For the wandboard-quad, we find the following directories under tmp/deploy/sources:

- ▶ allarch-poky-linux
- ▶ arm-poky-linux-gnueabi

And looking for what's distributed for the Linux kernel source, a GPLv2 package, we find under tmp/deploy/sources/arm-poky-linux-gnueabi/linux-wandboard-3.10.17-r0:

- ▶ defconfig
- ▶ github.com.wandboard-org.linux.git.tar.gz
- ▶ linux-wandboard-3.10.17-r0-recipe.tar.gz

So we have the kernel configuration, the source tarball, and the recipes used to build it, which include:

- ▶ linux-wandboard_3.10.17.bb
- ▶ linux-dtb.inc
- ▶ linux-wandboard.inc

And the license text for the root filesystem packages will also be included in the root filesystem under `/usr/share/common-licenses`, in a package directory hierarchy.

This configuration will provide deliverables for all build packages, but what we really want to do is provide them only for those whose licenses require us to.

For sure, we don't want to blindly distribute all the contents of the `sources` directory as is, as it will also contain our proprietary source, which we most likely don't want to distribute.

We can configure the `archiver` class only to provide the source for GPL and LGPL packages with the following:

```
COPyleft_License_Include = "GPL* LGPL*"  
COPyleft_License_Exclude = "CLOSED Proprietary"
```

And also, for an embedded product, we are usually only concerned with the software that ships in the product itself, so we can limit the recipe type to be archived to target images with the following:

```
Copyleft_Recipe_Types = "target"
```

We should obtain legal advice to decide which packages have licenses that make source distribution a requirement.

Other configuration options exist, such as providing the patched or configured source instead of the separated original source and patches, or source `rpms` instead of source tarballs. See the `archiver` class for more details.

There's more...

We can also choose to distribute the whole of our build environment. The best way to do this is usually to publish our BSP and software layers on a public Git repository. Our software layer can then provide `bblayers.conf.sample` and `local.conf.sample`, which can be used to set up ready-to-use build directories.

See also

- ▶ There are other requirements that haven't been discussed here, such as the mechanism chosen for distribution. It is recommended to get legal advice before releasing a product to ensure all the license obligations have been met.

Analyzing your system for compliance

The Yocto build system makes it easy to provide auditing information to our legal advisers. This recipe will explain how.

How to do it...

Under `tmp/deploy/licenses`, we find a directory list of packages (including their corresponding licenses) and an `image` folder with a package and license manifest.

For the example image provided before, `core-image-small`, we have the following:

```
tmp/deploy/licenses/core-image-small-wandboard-quad-<timestamp>/  
package.manifest  
base-files  
base-passwd  
busybox  
busybox-syslog  
busybox-udhcpc  
initscripts  
initscripts-functions  
libc6  
run-postinsts  
sysvinit  
sysvinit-inittab  
sysvinit-pidof  
update-alternatives-opkg  
update-rc.d
```

And the corresponding `tmp/deploy/licenses/core-image-small-wandboard-quad-<timestamp>/license.manifest` file excerpt is as follows:

```
PACKAGE NAME: base-files  
PACKAGE VERSION: 3.0.14  
RECIPE NAME: base-files  
LICENSE: GPLv2  
  
PACKAGE NAME: base-passwd  
PACKAGE VERSION: 3.5.29  
RECIPE NAME: base-passwd  
LICENSE: GPLv2+
```

These files can be used to analyze all the different packages that form our root filesystem. We can also audit them to make sure we comply with the licenses when releasing our product to the public.

There's more

You can instruct the Yocto build system to specifically avoid certain licenses by using the INCOMPATIBLE_LICENSE configuration variable. The usual way to use it is to avoid GPLv3-type licenses by adding the following to your conf/local.conf file:

```
INCOMPATIBLE_LICENSE = "GPL-3.0 LGPL-3.0 AGPL-3.0"
```

This will build core-image-minimal and core-image-base images as long as no extra image features are included.

Working with open source and proprietary code

It is common for an embedded product to be built upon an open source system like the one built by Yocto, and to include proprietary software that adds value and specializes the product. This proprietary part usually is intellectual property and needs to be protected, and it's important to understand how it can coexist with open source.

This recipe will discuss some examples of open source packages commonly found on embedded products and will briefly explain how to use proprietary software with them.

How to do it...

Open source licenses can be broadly divided into two categories based on whether they are:

- ▶ **Permissive:** These are similar to **Internet Software Consortium (ISC)**, MIT, and BSD licenses. They have few requirements attached to them and just require us to preserve copyright notices.
- ▶ **Restrictive:** These are similar to the GPL, which bind us to not only distribute the source code and modifications, either with the binary itself or at a later date, but also to distribute tools to build, install, and run the source.

However, some licenses might "pollute" modifications and derivative work with their own conditions, commonly referred to as *viral licenses*, while others will not. For example, if you link your application to GPL-licensed code, your application will be bound by the GPL too.

The virulent nature of the GPL has made some people wary of using GPL-licensed software, but it's important to note that proprietary software can run alongside GPL software as long as the license terms are understood and respected.

For example, violating the GPLv2 license would mean losing the right to distribute the GPLv2 code in the future, even if further distribution is GPLv2 compliant. In this case, the only way to be able to distribute the code again would be to ask the copyright holder for permission.

How it works...

Next, we will provide guidance regarding licensing requirements for some open source packages commonly used in embedded products. It does not constitute legal advice, and as stated before, proper legal auditing of your product should be done before public release.

The U-Boot bootloader

U-Boot is licensed under the GPLv2, but any program launched by it does not inherit its license. So you are free to use U-Boot to launch a proprietary operating system, for example. However, your final product must comply with the GPLv2 with regards to U-Boot, so U-Boot source code and modifications must be provided.

The Linux kernel

The Linux kernel is also licensed under the GPLv2. Any application that runs in the Linux kernel user space does not inherit its license, so you can run your proprietary software in Linux freely. However, Linux kernel modules are part of the Linux kernel and as such must comply with the GPLv2. Also, your final product must release the Linux kernel source and modifications, including external modules that run in your product.

Glibc

The GNU C library is licensed under the **Lesser General Public License (LGPL)**, which allows dynamic linking without license inheritance. So your proprietary code can dynamically link with `glibc`, but of course you still have to comply with the LGPL with regards to `glibc`. Note, however, that statically linking your application would pollute it with the LGPL.

BusyBox

BusyBox is also licensed under the GPLv2. The license allows for non-related software to run alongside it, so your proprietary software can run alongside BusyBox freely. As before, you have to comply with the GPLv2 with regards to BusyBox and distribute its source and modifications.

The Qt framework

Qt is licensed under three different licenses, which is common for open source projects. You can choose whether you want a commercial license (in which case, your proprietary application is protected), a LGPL license (which, as discussed before, would also protect your proprietary software by allowing the dynamic linking of your application as long as you complied with the LGPL for the Qt framework itself), or the GPLv3 (which would be inherited by your application).

The X Windows system

The X.Org source is licensed under permissive MIT-style licenses. As such, your proprietary software is free to make any use of it as long as its use is stated and copyright notices are preserved.

There's more...

Let's see how to integrate our proprietary-licensed code into the Yocto build system. When preparing the recipe for our application, we can take several approaches to licensing:

- ▶ Mark LICENSE as closed. This is the usual case for a proprietary application. We use the following:

```
LICENSE = "CLOSED"
```

- ▶ Mark LICENSE as proprietary and include some type of license agreement. This is commonly done when releasing binaries with some sort of end user agreement that is referenced in the recipe. For example, `meta-fsl-arm` uses this type of license to comply with Freescale's End User License Agreement. An example follows:

```
LICENSE = "Proprietary"  
LIC_FILES_CHKSUM = "file://EULA.txt;md5=93b784b1c11b3ffffb1638498  
a8dde3f6"
```

- ▶ Provide multiple licensing options, such as an open source license and a commercial license. In this case, the LICENSE variable is used to specify the open licenses, and the LICENSE_FLAGS variable is used for the commercial licenses. A typical example is the `gst-plugins-ugly` package in Poky:

```
LICENSE = "GPLv2+ & LGPLv2.1+ & LGPLv2+"  
LICENSE_FLAGS = "commercial"  
LIC_FILES_CHKSUM =  
    "file://COPYING;md5=a6f89e2100d9b6cdffcea4f398e37343 \  
    file://gst/synaesthesia/synaescope.h;beginline=1;endline=20  
    ;md5=99f301df7b80490c6ff8305fcc712838 \  
    file://tests/check/elements/xingmux.c;beginline=1;endline=2  
    1;md5=4c771b8af188724855cb99cadd390068 \  
    file://gst/mpegstream/gstmpegparse.h;beginline=1;endline=18  
    ;md5=ff65467b0c53cdffa98d0684c1bc240a9"
```

When the `LICENSE_FLAGS` variable is set on a recipe, the package will not be built unless the license appears on the `LICENSE_FLAGS_WHITELIST` variable too, typically defined in your `conf/local.conf` file. For the earlier example, we would add:

```
LICENSE_FLAGS_WHITELIST = "commercial"
```

The `LICENSE` and `LICENSE_FLAGS_WHITELIST` variables can match exactly for a very narrow match or broadly, as in the preceding example, which matches all licenses that begin with the word `commercial`. For narrow matches, the package name must be appended to the license name; for instance, if we only wanted to whitelist the `gst-plugins-ugly` package from the earlier example but nothing else, we could use the following:

```
LICENSE_FLAGS_WHITELIST = "commercial_gst-plugins-ugly"
```

See also

- ▶ You should refer to the specific licenses for a complete understanding of the requirements imposed by them. You can find a complete list of open source licenses and their documentation at <http://spdx.org/licenses/>.

4

Application Development

In this chapter, we will cover the following recipes:

- ▶ Preparing and using an SDK
- ▶ Using the Application Development Toolkit
- ▶ Using the Eclipse IDE
- ▶ Developing GTK+ applications
- ▶ Using the Qt Creator IDE
- ▶ Developing Qt applications
- ▶ Describing workflows for application development
- ▶ Working with GNU make
- ▶ Working with the GNU build system
- ▶ Working with the CMake build system
- ▶ Working with the SCons builder
- ▶ Developing with libraries
- ▶ Working with the Linux framebuffer
- ▶ Using the X Windows system
- ▶ Using Wayland
- ▶ Adding Python applications
- ▶ Integrating the Oracle Java Runtime Environment
- ▶ Integrating the Open Java Development Kit
- ▶ Integrating Java applications

Introduction

Dedicated applications are what define an embedded product, and Yocto offers helpful application development tools as well as the functionality to integrate with popular **Integrated Development Environments (IDE)** like Eclipse and Qt Creator. It also provides a wide range of utility classes to help in the integration of finished applications into the build system and the target images.

This chapter will introduce the IDEs and show us how they are used to build and debug C and C++ applications on real hardware, and will explore application development, including graphical frameworks and Yocto integration, not only for C and C++ but also Python and Java applications.

Preparing and using an SDK

The Yocto build system can be used to generate a cross-compilation toolchain and matching sysroot for a target system.

Getting ready

We will use the previously used wandboard-quad build directory and source the setup-environment script as follows:

```
$ cd /opt/yocto/fsl-community-bsp/  
$ source setup-environment wandboard-quad
```

How to do it...

There are several ways to build an SDK with the Yocto build system:

- ▶ The meta-toolchain target.

This method will build a toolchain that matches your target platform, and a basic sysroot that will not match your target root filesystem. However, this toolchain can be used to build bare metal software like the U-Boot bootloader or the Linux kernel, which do not need a sysroot. The Yocto project offers downloadable sysroot for the supported hardware platforms. You can also build this toolchain yourself with:

```
$ bitbake meta-toolchain
```

Once built, it can be installed with:

```
$ cd tmp/deploy/sdk  
$ ./poky-glibc-x86_64-meta-toolchain-cortexa9hf-vfp-neon-  
  toolchain-1.7.1.sh
```

- ▶ The `populate_sdk` task.

This is the recommended way to build a toolchain matching your target platform with a sysroot matching your target root filesystem. You build it with:

```
$ bitbake core-image-sato -c populate_sdk
```

You should replace `core-image-sato` for the target root filesystem image you want the sysroot to match. The resulting toolchain can be installed with:

```
$ cd tmp/deploy/sdk  
$ ./poky-glibc-x86_64-core-image-sato-cortexa9hf-vfp-neon-  
  toolchain-1.7.1.sh
```

Also, if you want your toolchain to be able to build static applications, you need to add static libraries to it. You can do this by adding specific static libraries to your target image, which could also be used for native compilation. For example, to add the static glibc libraries, add the following to your `conf/local.conf` file:

```
IMAGE_INSTALL_append = " glibc-staticdev"
```

And then build the toolchain to match your root filesystem as explained previously.

You usually won't want the static libraries added to your image, but do you want to be able to cross-compile static applications, so you can also add all the static libraries to the toolchain by adding:

```
SDKIMAGE_FEATURES_append = " staticdev-pkgs"
```

- ▶ The `meta-toolchain-qt` target.

This method will extend `meta-toolchain` to build Qt applications. We will see how to build Qt applications later on. To build this toolchain, execute the following command:

```
$ bitbake meta-toolchain-qt
```

Once built, it can be installed with:

```
$ cd tmp/deploy/sdk  
$ ./poky-glibc-x86_64-meta-toolchain-qt-cortexa9hf-vfp-neon-  
  toolchain-qt-1.7.1.sh
```

The resulting toolchain installers will be located under `tmp/deploy/sdk` for all the cases mentioned here.

- ▶ The `meta-ide-support` target.

This method does not generate a toolchain installer, but it prepares the current build project to use its own toolchain. It will generate an `environment-setup` script inside the tmp directory.

```
$ bitbake meta-ide-support
```

To use the bundled toolchain, you can now source that script as follows:

```
$ source tmp/environment-setup-cortexa9hf-vfp-neon-poky-linux-gnueabi
```

Using the Application Development Toolkit

The ADT is an SDK installation script that installs the following for Poky-supported hardware platforms:

- ▶ A prebuilt cross-compilation toolchain, as explained previously
- ▶ A sysroot that matches the `core-image-sato` target image
- ▶ The QEMU emulator
- ▶ Other development user space tools used for system profiling (these will be discussed in the following chapters)

Getting ready

To install the ADT, you can choose either of the following options:

- ▶ Download a precompiled tarball from the Yocto project downloads site with the following command:

```
$ wget http://downloads.yoctoproject.org/releases/yocto/yocto-1.7.1/adt-installer/adt_installer.tar.bz2
```
- ▶ Build one using your Yocto build directory.

The ADT installer is an automated script to install precompiled Yocto SDK components, so it will be the same whether you download the prebuilt version or you build one yourself.

You can then configure it before running it to customize the installation.

Note that it only makes sense to use the ADT for the Poky-supported platforms. For instance, it is not that useful for external hardware like `wandboard-quad` unless you provide your own components.

How to do it...

To build the ADT from your Yocto build directory, open a new shell and execute the following:

```
$ cd /opt/yocto/poky  
$ source oe-init-build-env qemuarm  
$ bitbake adt-installer
```

The ADT tarball will be located in the `tmp/deploy/sdk` directory.

How it works...

To install it, follow these steps:

1. Extract the tarball on a location of your choice:

```
$ cd /opt/yocto  
$ cp  
  /opt/yocto/poky/qemuarm/tmp/deploy/sdk/adt_installer.tar.bz2  
  /opt/yocto  
$ tar xvf adt_installer.tar.bz2  
$ cd /opt/yocto/adt-installer
```

2. Configure the installation by editing the `adt_installer.conf` file. Some of the options are:

- ❑ `YOCTOADT_REPO`: This is a repository with the packages and root filesystem to be used. By default, it uses the one on the Yocto project web site, <http://adtrepo.yoctoproject.org/1.7.1/>, but you could set one up yourself with your customized packages and root filesystem.
- ❑ `YOCTOADT_TARGETS`: This defines the machine targets the SDK is for. By default, this is ARM and x86.
- ❑ `YOCTOADT_QEMU`: This option controls whether to install the QEMU emulator. The default is to install it.
- ❑ `YOCTOADT_NFS_UTIL`: This option controls whether to install user mode NFS. It is recommended if you are going to use the Eclipse IDE with QEMU-based machines. The default is to install it.

And then for the specific target architectures (only shown for ARM):

- ❑ `YOCTOADT_ROOTFS_arm`: This defines the specific root filesystem images to download from the ADT repository. By default it installs the `minimal` and `sato-sdk` images.

- ❑ **YOCTOADT_TARGET_SYSROOT_IMAGE_arm:** This is the root filesystem used to create the sysroot. This must also be included in the **YOCTOADT_ROOTFS_arm** selection that was explained earlier. By default this is the **sato-sdk** image.
- ❑ **YOCTOADT_TARGET_MACHINE_arm:** This is the machine that the images are downloaded for. By default this is **qemuarm**.
- ❑ **YOCTOADT_TARGET_SYSROOT_LOC_arm:** This is the path on the host to install the target's sysroot. By default this is **\$HOME/test-yocto/**.

3. Run the ADT installer as follows:

```
$ ./adt_installer
```

It will ask for an installation location (by default **/opt/poky/1.7.1**) and whether you want to run it in interactive or silent mode.

Using the Eclipse IDE

Eclipse is an open source IDE that is written mostly in Java and released under the **Eclipse Public License (EPL)**. It can be extended using plugins, and the Yocto project releases a Yocto plugin that allows us to use Eclipse for application development.

Getting ready

Yocto 1.7 provides Eclipse Yocto plugins for two different Eclipse versions, Juno and Kepler. They can be downloaded at <http://downloads.yoctoproject.org/releases/yocto/yocto-1.7.1/eclipse-plugin/>. We will use Kepler 4.3, as it is the newest. We will start with the Eclipse Kepler standard edition and install all the required plugins we need.

It is recommended to run Eclipse under Oracle Java 1.7, although other Java providers are supported. You can install Oracle Java 1.7 from Oracle's web site, <https://www.java.com/en/>, or using a Ubuntu Java Installer PPA, <https://launchpad.net/~webupd8team/+archive/ubuntu/java>. The latter will integrate Java with your package management system, so it's preferred. To install it, follow these steps:

```
$ sudo add-apt-repository ppa:webupd8team/java  
$ sudo apt-get update  
$ sudo apt-get install oracle-java7-set-default
```

To download and install Eclipse Kepler standard edition for an **x86_64** host, follow these steps:

1. Fetch the tarball from the Eclipse download site, <http://eclipse.org/downloads/packages/release/Kepler/SR2>. For example:

```
$ wget http://download.eclipse.org/technology/epp/downloads/
```

```
release/kepler/SR2/eclipse-standard-kepler-SR2-linux-gtk-x86_64.  
tar.gz
```

2. Unpack it on a location of your choice as follows:

```
$ tar xvf eclipse-standard-kepler-SR2-linux-gtk-x86_64.tar.gz
```

3. Start the Eclipse IDE with the following:

```
$ nohup eclipse &
```

4. Select **Install New Software** from the **Help** pull-down menu. Then select the **Kepler - http://download.eclipse.org/releases/kepler** source.
5. Install the following Eclipse components:

- Linux tools:

LTTng - Linux Tracing Toolkit

- Mobile and device development:

C/C++ Remote Launch

Remote System Explorer End-user Runtime

Remote System Explorer User Actions

Target Management Terminal

TCF Remote System Explorer add-in

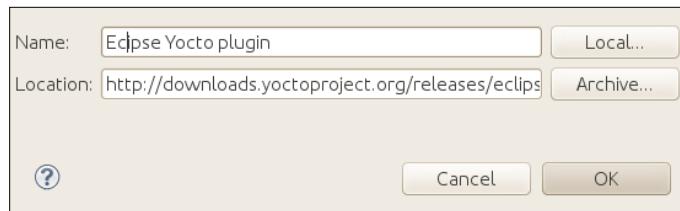
TCF Target Explorer

- Programming languages:

C/C++ Autotools Support

C/C++ Development Tools

6. Install the Eclipse Yocto plugin by adding this repository source: `http://downloads.yoctoproject.org/releases/eclipse-plugin/1.7.1/kepler`, as shown in the following screenshot:



7. Choose **Yocto Project ADT plug-in** and ignore the unsigned content warning. We won't be covering other plugin extensions.

How to do it...

To configure Eclipse to use a Yocto toolchain, go to **Window | Preferences | Yocto Project ADT**.

The ADT configuration offers two cross-compiler options:

1. **Standalone pre-built toolchain:** Choose this when you have installed a toolchain either from a toolchain installer or the ADT installer.
2. **Build system derived toolchain:** Choose this when using a Yocto build directory prepared with `meta-ide-support` as explained previously.

It also offers two target options:

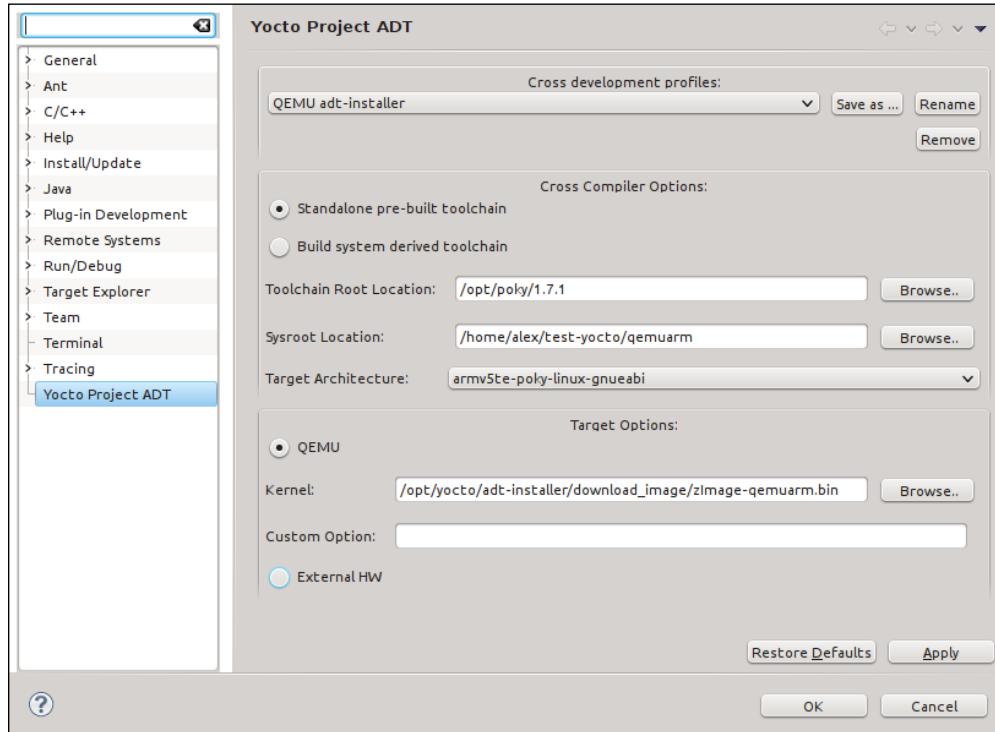
1. **The QEMU emulator:** Choose this if you are using Poky with a virtualized machine and you have used the ADT installer to install a qemux86 Linux kernel and root filesystem.
2. **External hardware:** Choose this if you are using real hardware like the wandboard-quad hardware. This option is the most useful for embedded development.

An example configuration when using the ADT installer with its default configuration would be to choose the standalone prebuilt toolchain option along with the QEMU emulator as follows:

- ▶ Cross-compiler options:
 - Standalone pre-built toolchain:
Toolchain root location: /opt/poky/1.7.1
Sysroot location: \${HOME}/test-yocto/qemux86
Target architecture: armv5te-poky-linux-gnueabi

- Target options:

QEMU kernel: /tmp/adt-installer/download_image/zImage-qemuarm.bin



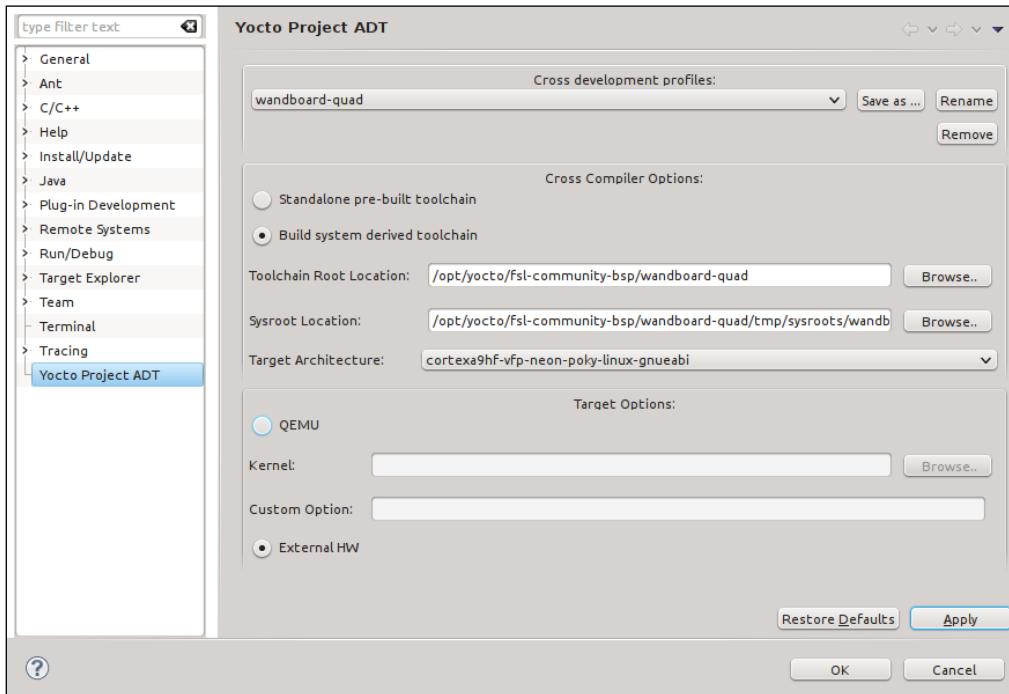
And for a build system derived toolchain using the wandboard-quad reference board, this is what you will need:

- ▶ Cross-compiler options:

- Build system derived toolchain:

Toolchain root location: /opt/yocto/fsl-community-bsp/wandboard-quad

Sysroot location: /opt/yocto/fsl-community-bsp/wandboard-quad/tmp/sysroots/wandboard-quad



There's more...

In order to perform debugging on a remote target, it needs to be running the `tcf-agent` daemon. It is included by default on the SDK images, but you can also include it in any other image by adding the following to your `conf/local.conf` file:

```
EXTRA_IMAGE_FEATURES += "eclipse-debug"
```

See also

- ▶ For more information, refer to the *Yocto Project Application Developer's Guide* at <http://www.yoctoproject.org/docs/1.7.1/adt-manual/adt-manual.html>

Developing GTK+ applications

This recipe will show how to build, run, and debug a graphical GTK+ application using the Eclipse IDE.

Getting ready

1. Add the `eclipse-debug` feature to your project's `conf/local.conf` file as follows:

```
EXTRA_IMAGE_FEATURES += "eclipse-debug"
```

2. Build a `core-image-sato` target image as follows:

```
$ cd /opt/yocto/fsl-community-bsp/  
$ source setup-environment wandboard-quad  
$ bitbake core-image-sato
```

3. Build a `core-image-sato` toolchain as follows:

```
$ bitbake -c populate_sdk core-image-sato
```

4. Install the toolchain as follows:

```
$ cd tmp/deploy/sdk  
$ ./poky-glibc-x86_64-core-image-sato-cortexa9hf-vfp-neon-  
toolchain-1.7.1.sh
```

Before launching the Eclipse IDE, we can check whether we are able to build and launch a GTK application manually. We will build the following GTK+ hello world application:

The following is a code for `gtk_hello_world.c`:

```
#include <gtk/gtk.h>  
  
int main(int argc, char *argv[])  
{  
    GtkWidget *window;  
    gtk_init (&argc, &argv);  
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);  
    gtk_widget_show (window);  
    gtk_main ();  
    return 0;  
}
```

Application Development

To build it, we use the `core-image-sato` toolchain installed as described previously:

```
$ source /opt/poky/1.7.1/environment-setup-cortexa9hf-vfp-neon-poky-linux-gnueabi
$ ${CC} gtk_hello_world.c -o helloworld `pkg-config --cflags --libs gtk+-2.0`
```

This command uses the `pkg-config` helper tool to read the `.pc` files that are installed with the GTK libraries in the `sysroot` to determine which compiler switches (`--cflags` for include directories and `--libs` for the libraries to link with) are needed to compile programs that use GTK.

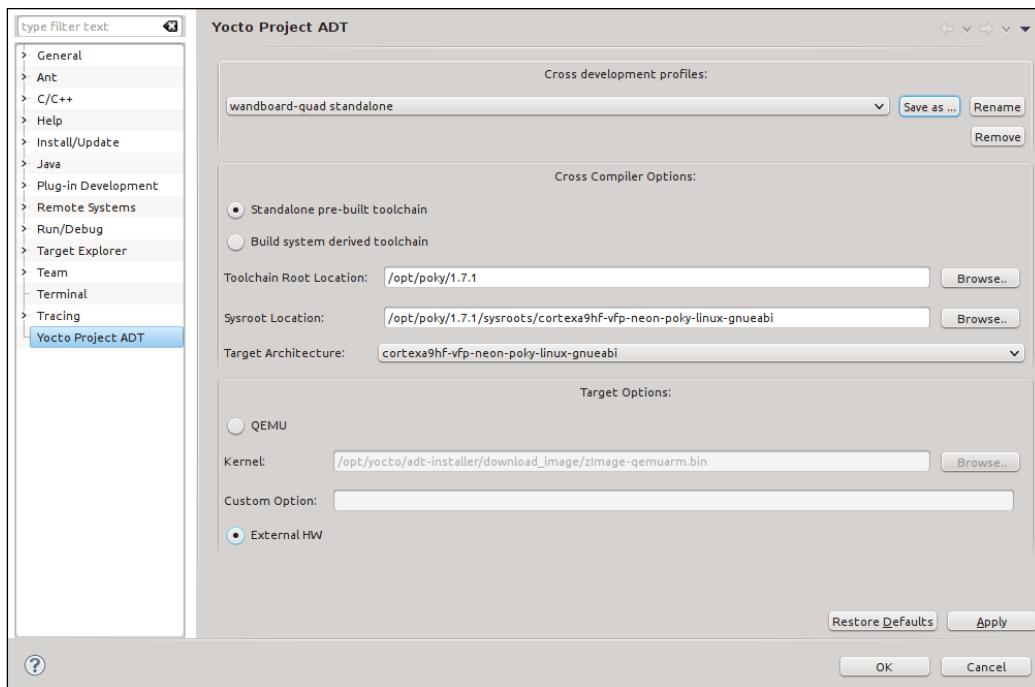
We can manually copy the resulting binary to our Wandboard while booting `core-image-sato` over NFS and run it from the target's console with:

```
# DISPLAY=:0 helloworld
```

This will open a GTK+ window over the SATO desktop.

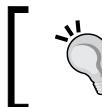
How to do it...

We can now configure the Eclipse ADT plugin using the standalone toolchain as described before, or we could decide to use the build system derived toolchain instead.



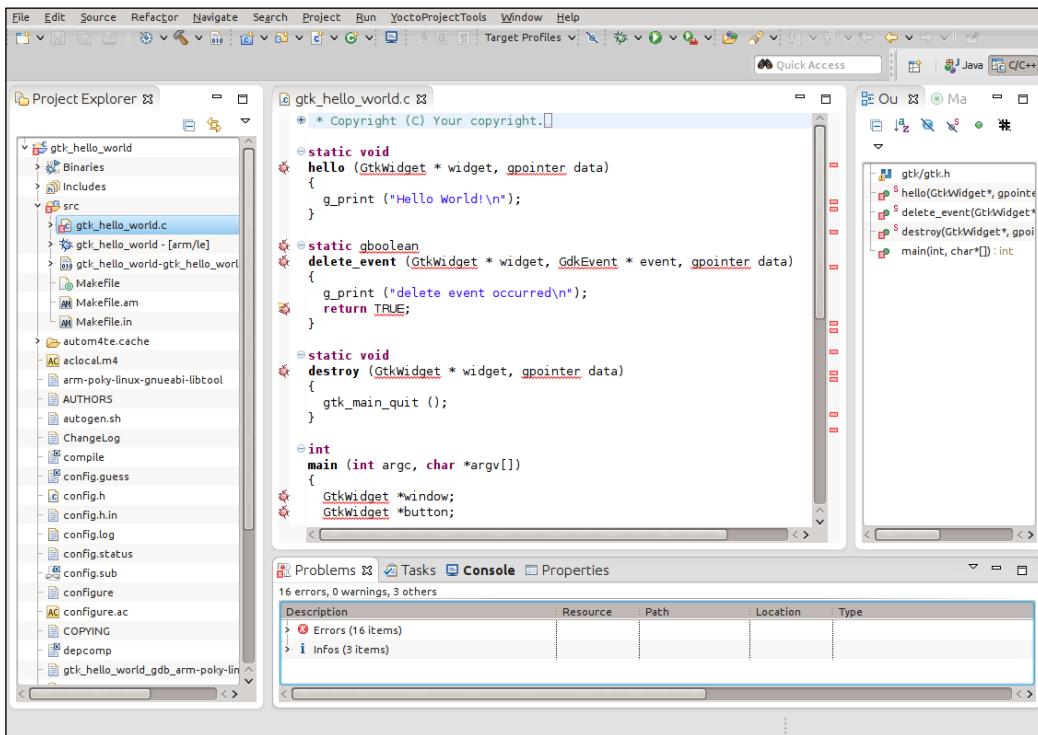
Follow the next steps to build and run an example hello world application:

1. Create a new hello world GTK autotools project. Accept all the defaults in the project creation wizard. Browse to **File | New | Project | C/C++ | C Project | Yocto Project ADT Autotools Project | Hello World GTK C Autotools Project**.



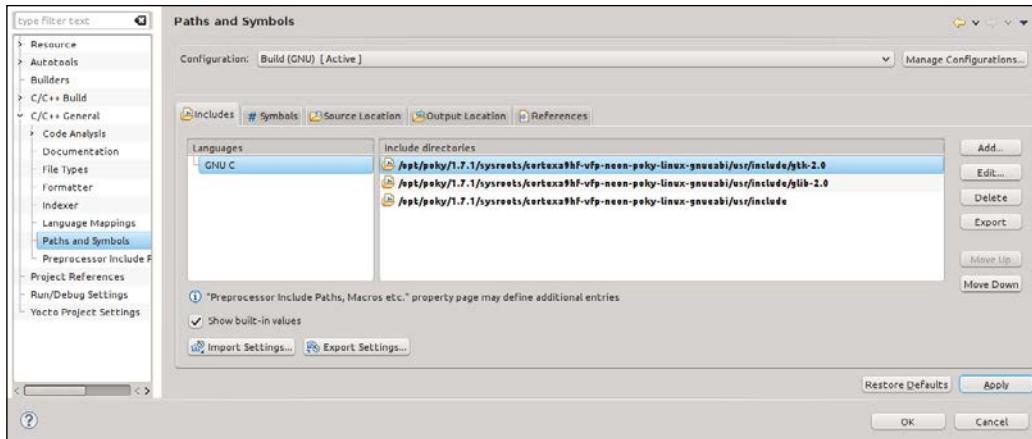
When choosing a name for your project, avoid using special characters like dashes, as they could cause problems with the build tools.

2. Build the project by going to **Project | Build Project**.

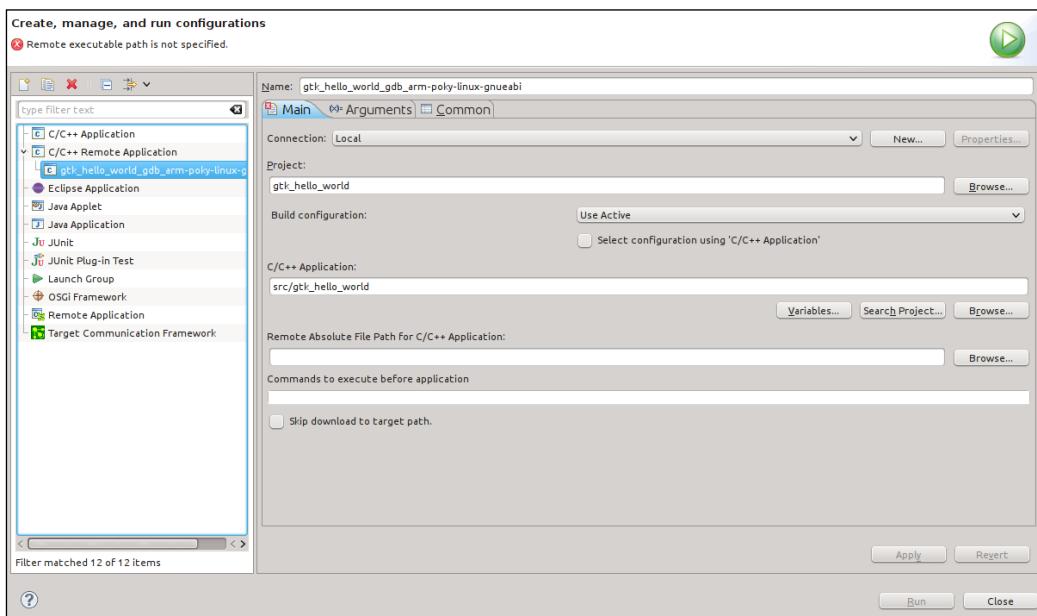


3. Even though the project builds successfully, you may see errors both marked in the source and in the **Problems** tab. This is because the Eclipse's code analysis feature cannot resolve all the project's symbols. To resolve it, add the needed include header files to your project's properties by going to **Project | Properties | C/C++ General | Paths and Symbols | Includes**.

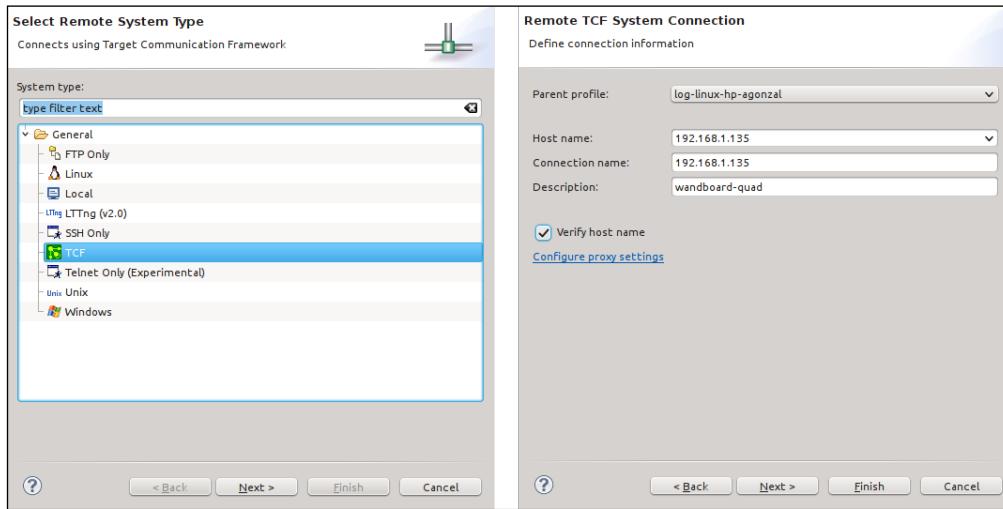
Application Development



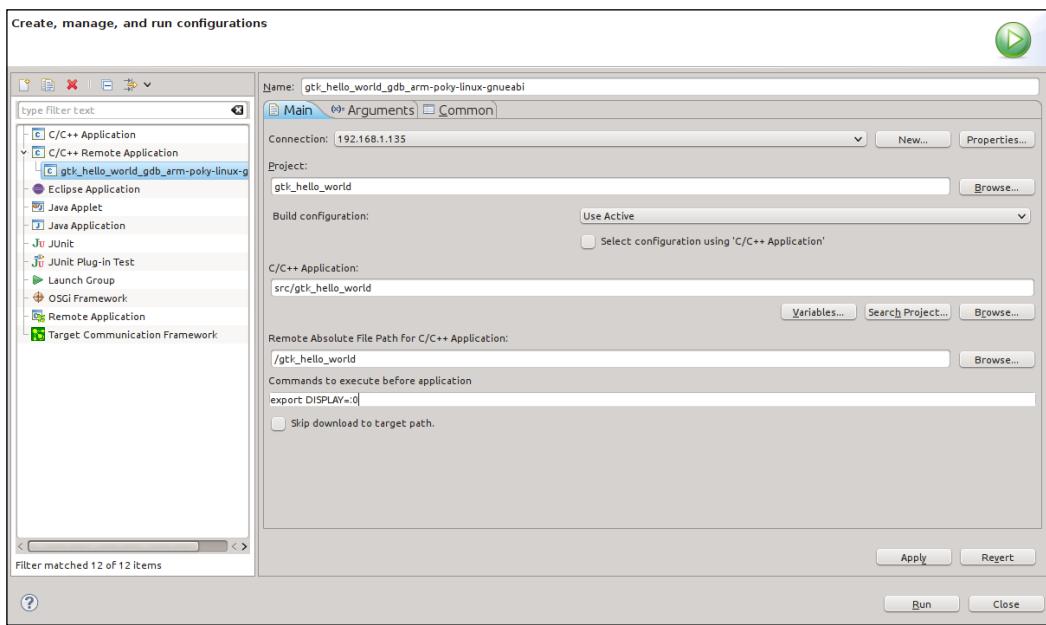
4. Under **Run | Run Configurations**, you should have **C/C++ Remote Application** with a TCF target called <project_name>_gdb_arm-poky-linux-gnueabi. If you don't, create one.



5. Create a new TCF connection to the target's IP address using the **New...** button in the **Main** tab.

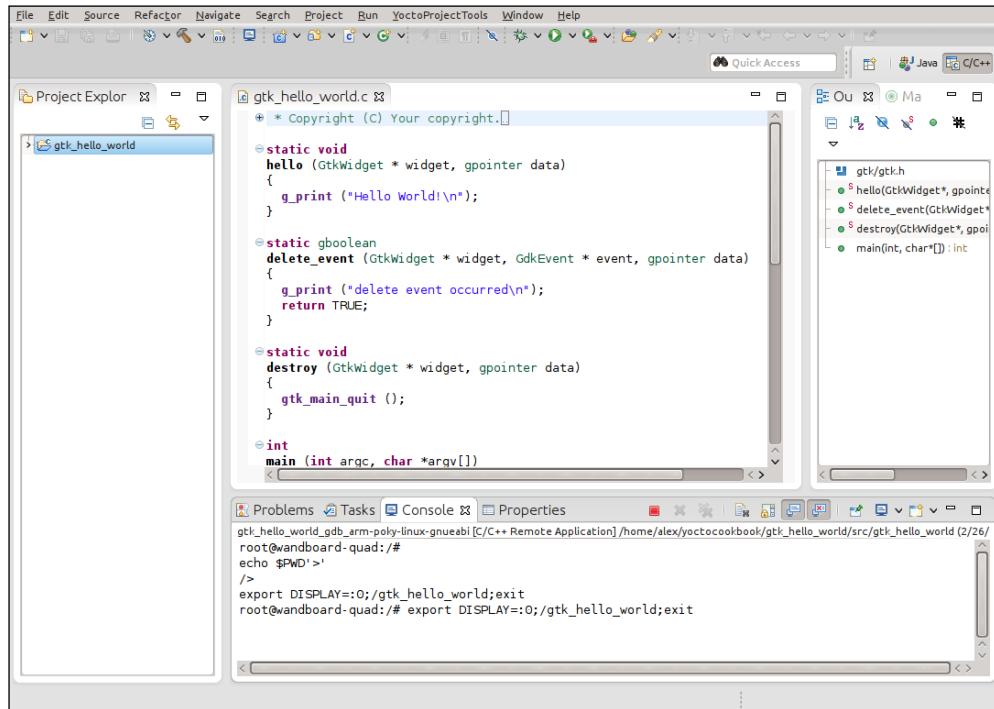


6. Fill in the **Remote Absolute File Path for C/C++ Application** field with the path to the binary and include the binary name; for example, /gtk_hello_world.
 7. In the **Commands to execute before application** field, enter `export DISPLAY=:0`.



Application Development

8. Run the application and log in as `root` with an empty password. You should see the GTK application on your SATO desktop, and the following output in the Console tab:



The screenshot shows the Eclipse IDE interface. The Project Explorer view on the left displays a single project named "gtk_hello_world". The Source Editor view in the center shows the code for "gtk_hello_world.c". The code defines a GTK application with a "hello" function that prints "Hello World!", a "delete_event" handler that prints "delete event occurred", a "destroy" handler that quits the application, and a "main" function that starts the application. The Console tab at the bottom shows the terminal output of the application being run as root on a remote target. The output includes commands like "echo \$PWD", "DISPLAY=:0; gtk_hello_world; exit", and "root@wandboard-quad:~#".

If you have problems connecting to the target, verify that it is running `tcf-agent` by typing in the following on the target's console:

 # ps w | grep tcf
735 root 11428 S /usr/sbin/tcf-agent -d -L-
-10

If you have login problems, you can use Eclipse's **Remote System Explorer (RSE)** perspective to clear passwords and debug the connection to the target. Once the connection can be established and you are able to browse the target's filesystem through RSE, you can come back to the run configuration.

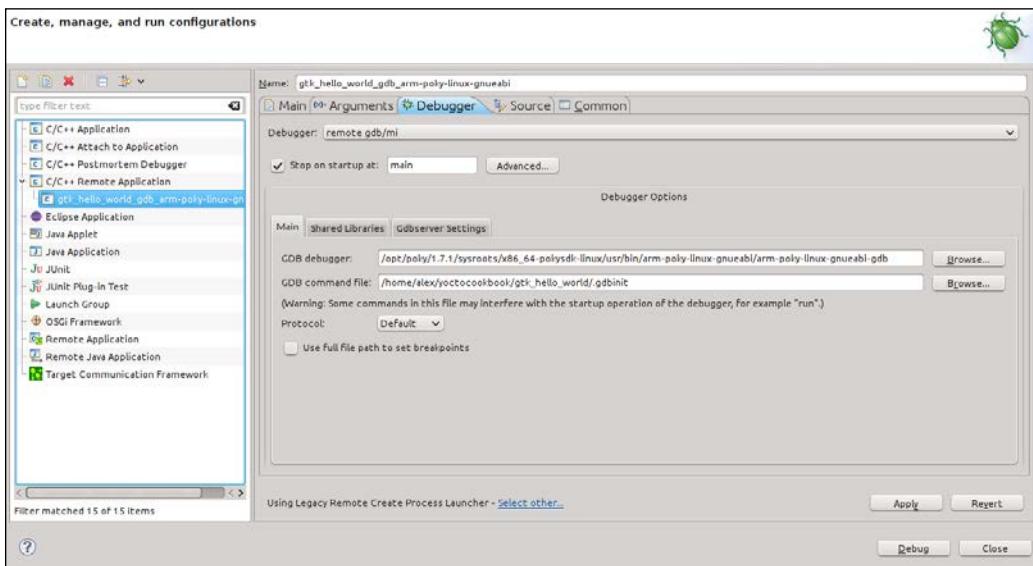
There's more...

To debug the application, follow these steps:

1. Go to **Run | Debug Configuration**.
2. Under the **Debugger** tab, verify the GDB debugger path is the correct toolchain debugger location.

```
/opt/poky/1.7.1/sysroots/x86_64-pokysdk-linux/usr/bin/arm-poky-linux-gnueabi/arm-poky-linux-gnueabi-gdb
```

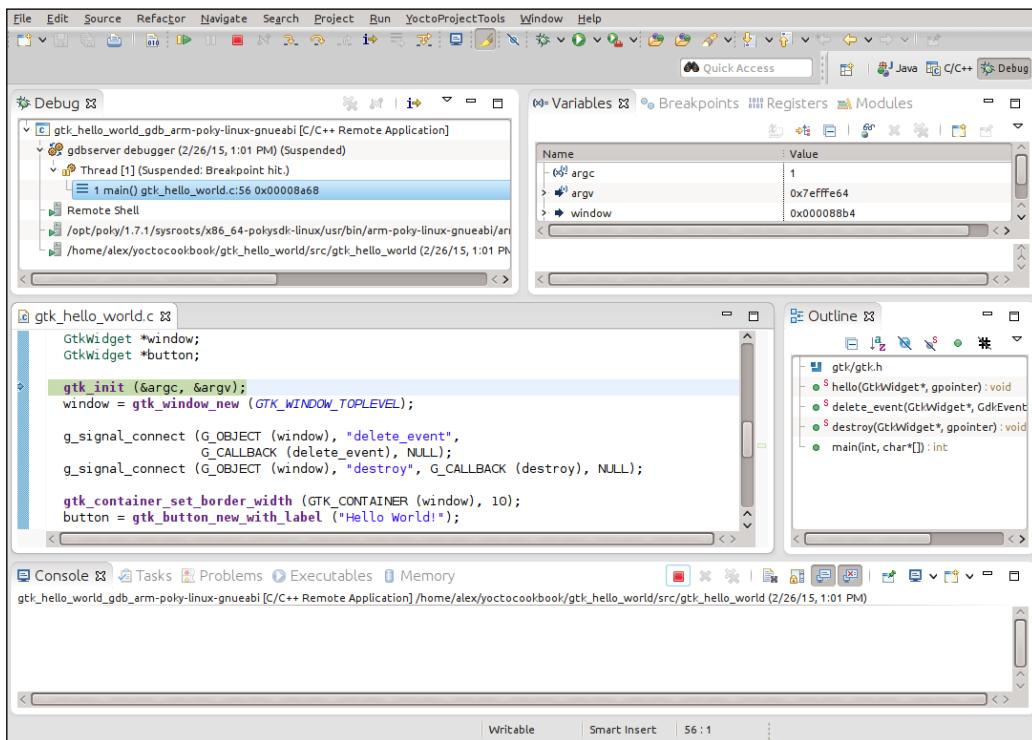
If it isn't, point it to the correct location.



3. Double-click on the `main` function in the source file to add a breakpoint. A blue dot will appear on the side bar.

Application Development

- Click on the **Debug** button. The debug perspective appears with the application executing on the remote Wandboard hardware.



If you get **Text file busy** error, remember to close the application we ran on the previous point.

Using the Qt Creator IDE

Qt Creator is a multiplatform IDE part of the Qt Application Development Framework SDK. It is the IDE of choice for Qt application development and is available with multiple licenses, including GPLv3, LGPLv2, and commercial licenses as well.

Getting ready

1. Download and install the Qt Creator 3.3.0 for your host from the Qt project downloads website. For downloading and installing an x86_64 Linux host, you can use the following commands:

```
$ wget  
http://download.qt.io/official_releases/qtcreator/3.3/3.3.0/qt  
-creator-opensource-linux-x86_64-3.3.0.run  
  
$ chmod u+x qt-creator-opensource-linux-x86_64-3.3.0.run  
  
$ ./qt-creator-opensource-linux-x86_64-3.3.0.run
```

2. Build a toolchain that is ready to develop Qt applications with the following:

```
$ cd /opt/yocto/fsl-community-bsp/  
  
$ source setup-environment wandboard-quad  
  
$ bitbake meta-toolchain-qt
```

3. Install it as follows:

```
$ cd tmp/deploy/sdk  
  
$ ./poky-glibc-x86_64-meta-toolchain-qt-cortexa9hf-vfp-neon-  
toolchain-qt-1.7.1.sh
```

How to do it...

Before launching Qt Creator, we need to set up the development environment. To make this happen automatically when we launch Qt Creator, we can patch its initialization script by adding the following line right at the beginning of the bin/qtcreator.sh file:

```
source /opt/poky/1.7.1/environment-setup-cortexa9hf-vfp-neon-poky-  
linux-gnueabi  
  
#! /bin/sh
```



Note that the environment initialization script is placed before the hash bang.

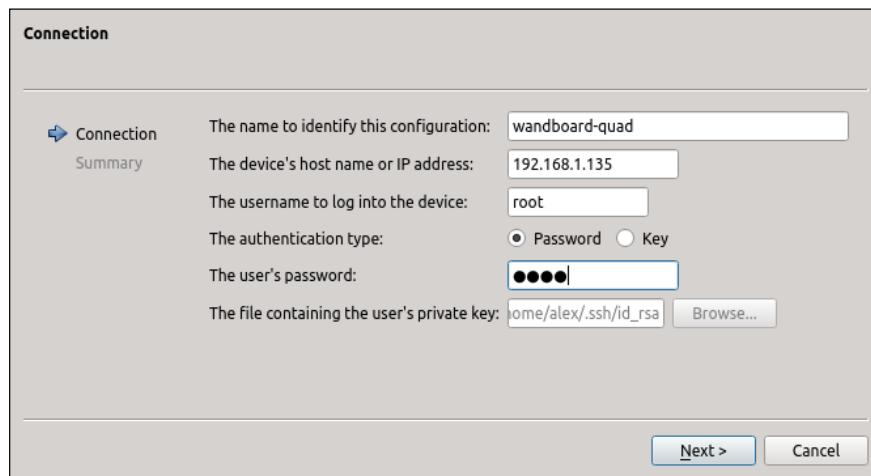
Now we can run Qt Creator as follows:

```
$ ./bin/qtcreator.sh &
```

Application Development

And configure it by going to **Tools | Options** and using the following steps:

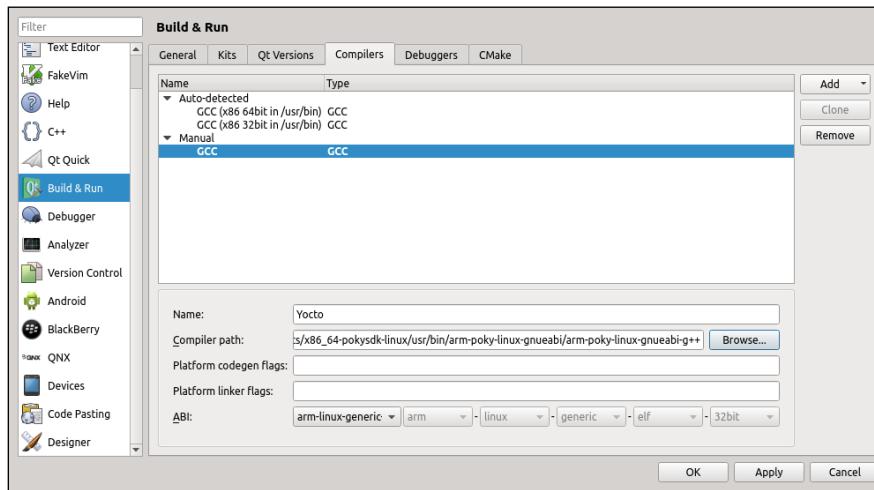
1. First we configure a new device for our Wandboard. Under **Devices | Add**, we select **Generic Linux Device**.



Set the root password in the target by using the `passwd` command from the target's root console and type it in the password field.

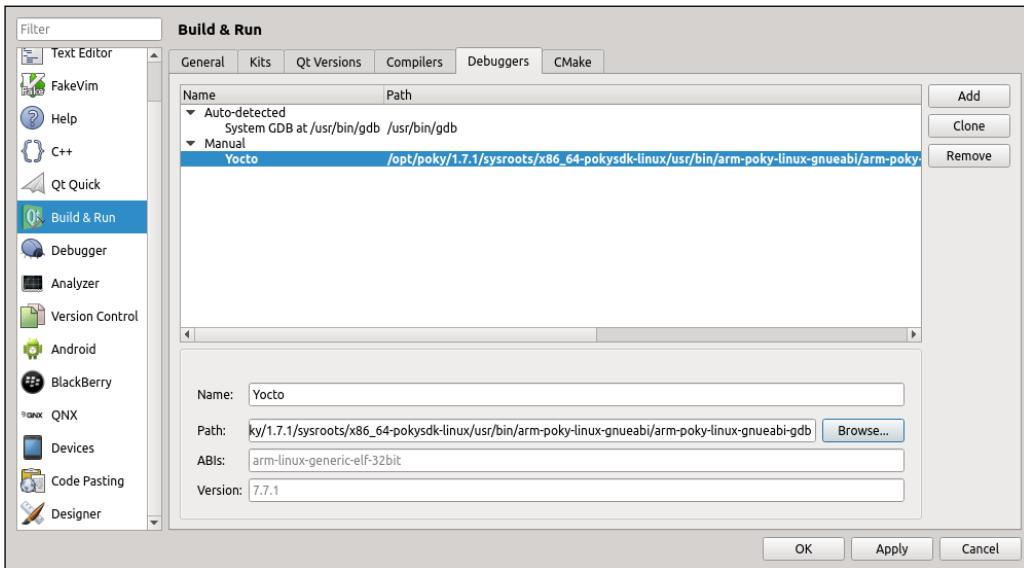
2. Under **Build & Run**, we configure a new compiler pointing to the Yocto meta-toolchain-qt compiler path we just installed. Here's the path as shown in the following screenshot:

```
/opt/poky/1.7.1/sysroots/x86_64-pokysdk-linux/usr/bin/arm-poky-linux-gnueabi/arm-poky-linux-gnueabi-g++
```



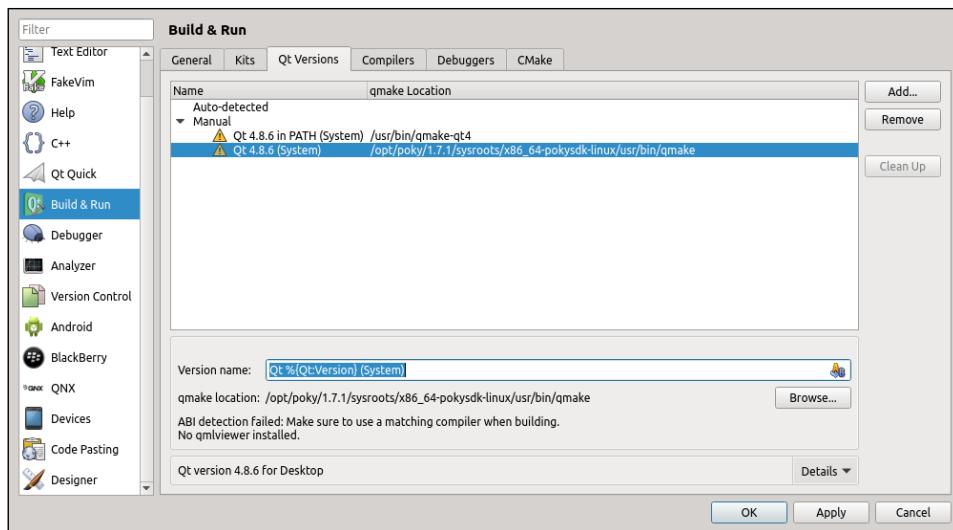
3. Similarly for a cross-debugger, the following is the path which is also mentioned in the following screenshot:

```
/opt/poky/1.7.1/sysroots/x86_64-pokysdk-linux/usr/bin/arm-poky-linux-gnueabi/arm-poky-linux-gnueabi-gdb
```



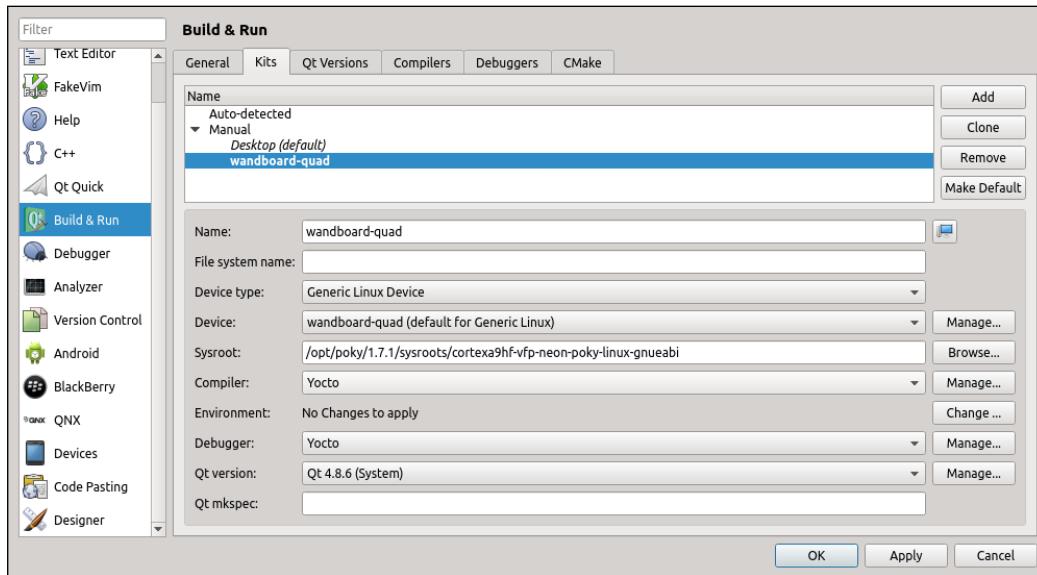
4. And then we configure Qt by selecting the qmake builder from the toolchain. Here's the path which is also mentioned in the following screenshot:

```
/opt/poky/1.7.1/sysroots/x86_64-pokysdk-linux/usr/bin/qmake
```



5. Finally we configure a new kit as follows:

1. Select **Generic Linux Device** and configure its sysroot to:
`/opt/poky/1.7.1/sysroots/cortexa9hf-vfp-neon-poky-linux-gnueabi/`
2. Select the compiler, debugger, and Qt version we just defined.



[ In Ubuntu, Qt Creator stores its configuration on the user's home directory under `.config/QtProject/`.]

Developing Qt applications

This recipe will show how to build, run, and debug a graphical Qt application using Qt Creator.

Getting ready

Before launching Qt Creator, we check whether we are able to build and launch a Qt application manually. We will build a Qt hello world application.

Here is the code for `qt_hello_world.cpp`:

```
#include <QApplication>
#include <QPushButton>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QPushButton hello("Hello world!");

    hello.show();
    return app.exec();
}
```

To build it, we use the `meta-toolchain-qt` installed as described previously:

```
$ source /opt/poky/1.7.1/environment-setup-cortexa9hf-vfp-neon-poky-
linux-gnueabi
$ qmake -project
$ qmake
$ make
```

This uses `qmake` to create a project file and a `Makefile` file with all the relevant code files in the folder.

To run it, we first need to build a filesystem with Qt support. We first prepare the environment as follows:

```
$ cd /opt/yocto/fsl-community-bsp/
$ source setup-environment wandboard-quad
```

And configure our project with the `qt4-pkgs` extra feature by adding the following to `conf/local.conf`:

```
EXTRA_IMAGE_FEATURES += "qt4-pkgs"
```

And for Qt applications, we also need the **International Component for Unicode (ICU)** library, as the Qt libraries are compiled with support for it.

```
IMAGE_INSTALL_append = " icu"
```

And build it with:

```
$ bitbake core-image-sato
```

Once finished, we can program the microSD card image and boot the Wandboard. Copy the `qt_hello_world` binary to the target and run:

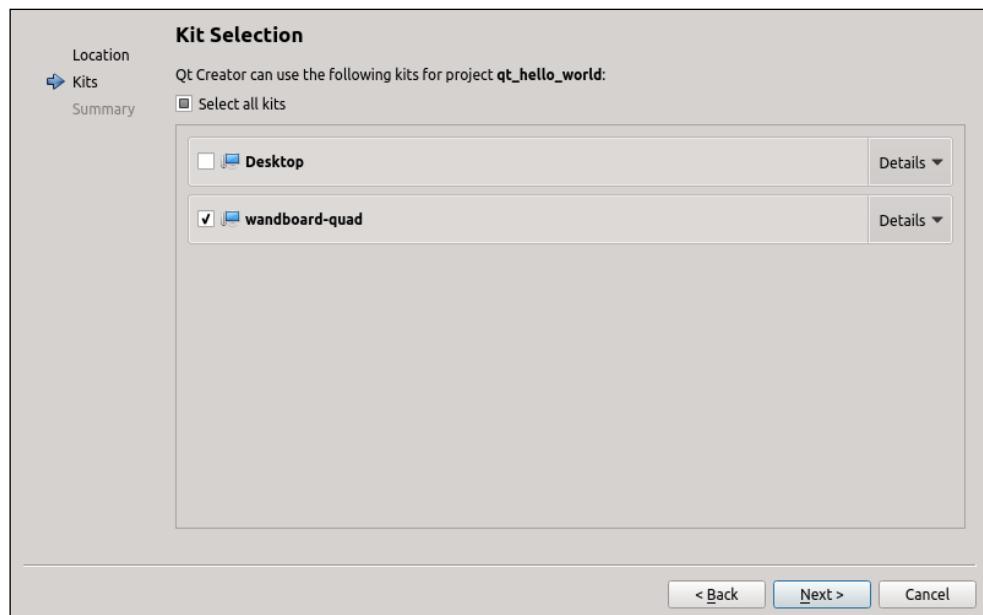
```
# DISPLAY=:0 qt_hello_world
```

You should see the Qt hello world window on the X11 desktop.

How to do it...

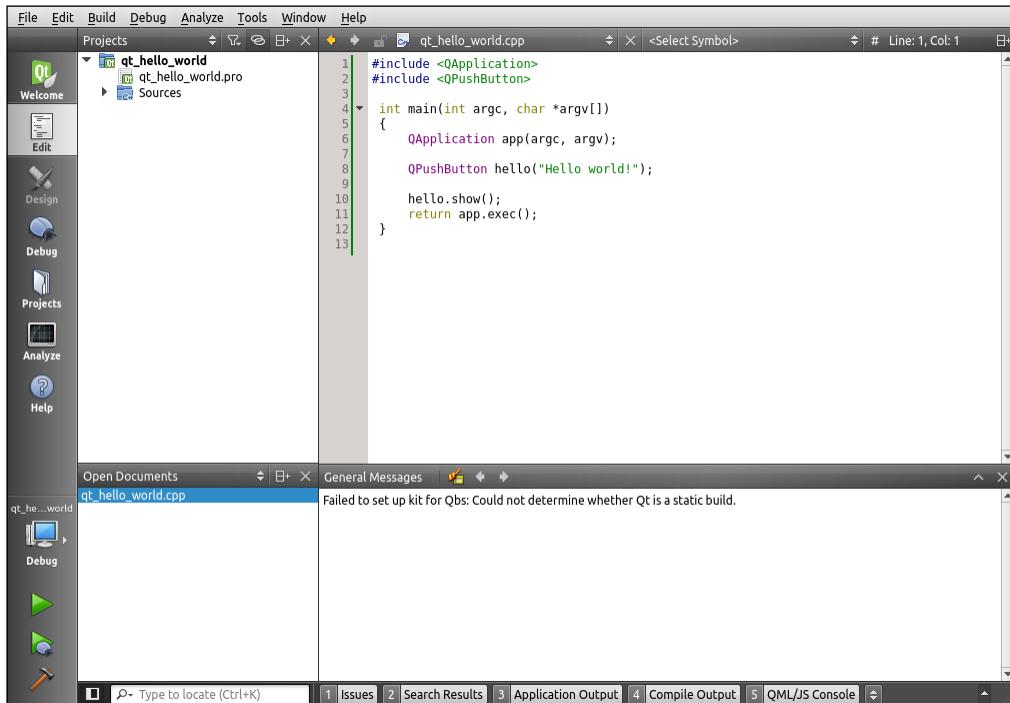
Follow these steps to build and run an example hello world application:

1. Create a new empty project by going to **File | New File or Project | Other project | Empty qmake project**.
2. Select only the **wandboard-quad** kit we just created.



3. Add a new C++ file, `qt_hello_world.cpp`, by going to **File | New File or Project | C++ | C++ Source File**.

4. Paste the contents of the `qt_hello_world.cpp` file into Qt Creator, as shown in the following screenshot:



5. Configure your project with the target installation details by adding the following to your `hw.pro` file:

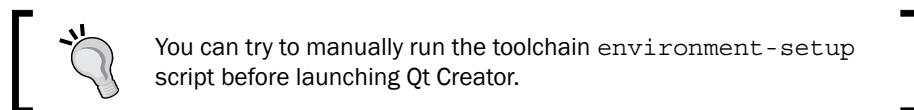
```
SOURCES += \
    qt_hello_world.cpp

TARGET = qt_hello_world
target.files = qt_hello_world
target.path = /
```

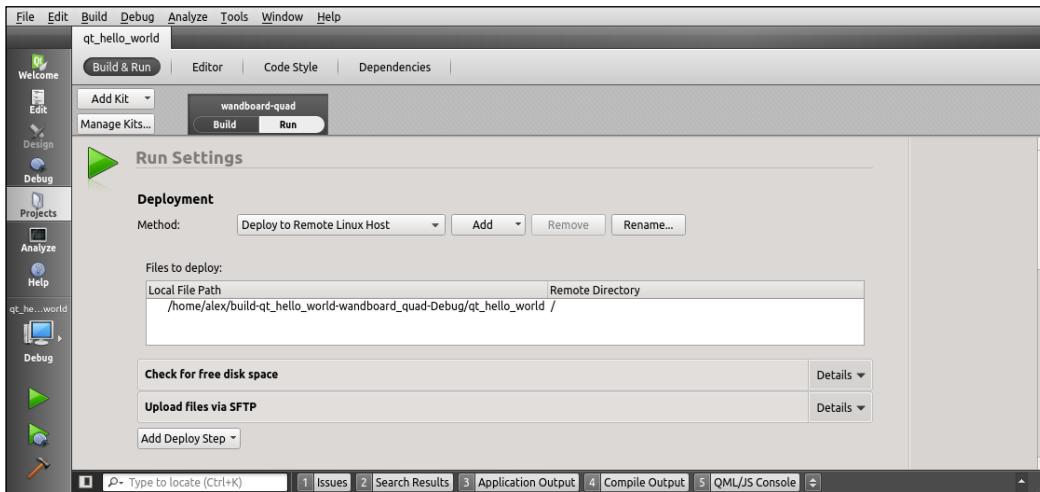
INSTALLS += target

Replace `qt_hello_world` with the name of your project.

6. Build the project. If you have build errors, verify that the Yocto build environment has been correctly set up.



7. Go to **Projects** | **Run** and check your project settings.



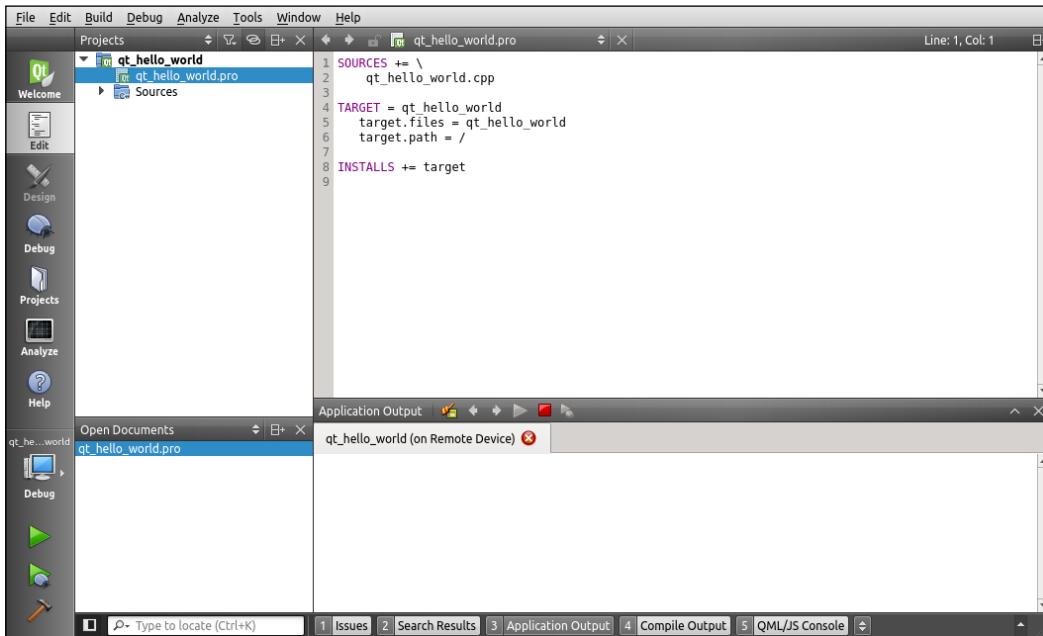
8. As can be seen in this screenshot, Qt Creator will use the SFTP protocol to transfer the files to the target. By default, the dropbear SSH server running on `core-image-sato` does not have SFTP support. We need to add it to our image to allow Qt Creator to work by adding the `openssh-sftp-server` package to the project's `conf/local.conf` file.

```
IMAGE_INSTALL_append = " openssh-sftp-server"
```

However, there are other tools we will need, like the **gdbserver** if we want to debug our application, so it's easier to add the `eclipse-debug` feature, which will add all of the needed applications to the target image.

```
EXTRA_IMAGE_FEATURES += "eclipse-debug"
```

9. You can now run the project.

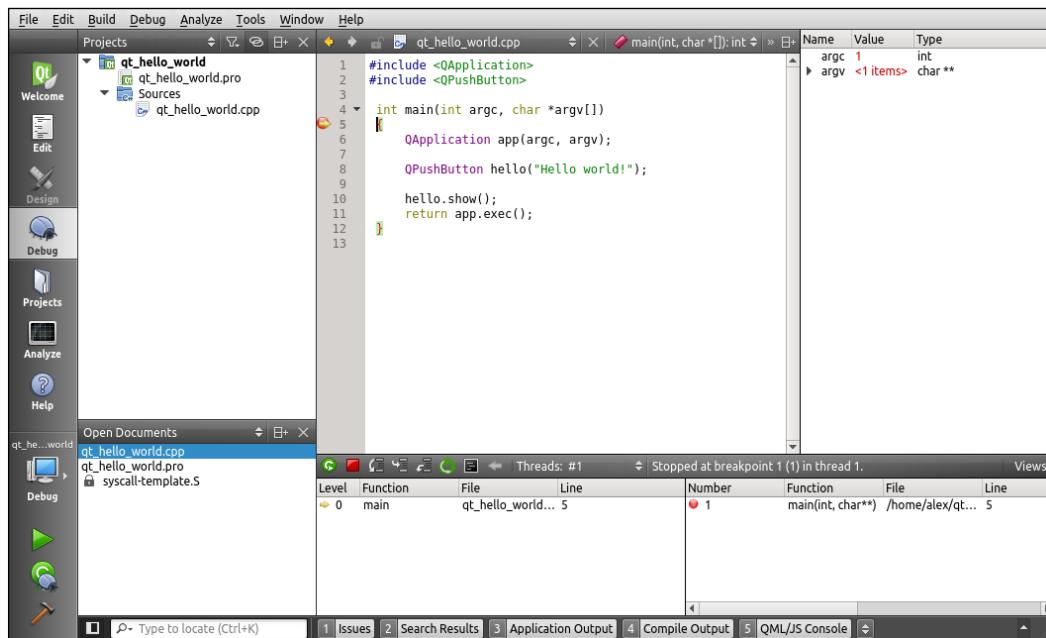


[ If the application fails to be deployed with a login error, verify that you have set a root password in the target as explained in the recipe previously, or that you are using SSH key authentication.]

You should now see the example Qt hello world application running on your SATO desktop.

There's more...

To debug the application, toggle a breakpoint on the source and click on the **Debug** button.



Describing workflows for application development

The workflows for application development are similar to the ones we already saw for U-Boot and the Linux kernel back in *Chapter 2, The BSP Layer*.

How to do it...

We will see how the following development workflows are applied to application development:

- ▶ External development
- ▶ Working directory development
- ▶ External source development

How it works...

External development

This is what we have been using on the recipes we saw before when building from the command line using a standalone toolchain, and also when using both the Eclipse and Qt Creator IDEs. This workflow produces binaries that have to be individually copied to the hardware to run and debug. It can be used in conjunction with the other workflows.

Working directory development

When the application is being built by the Yocto build system, we use this workflow to debug sporadic problems. However, it is not the recommended workflow for long developments. Note, though, that it is usually the first step when debugging third-party packages.

We will use the `helloworld_1.0.bb` custom recipe we saw back in the *Adding new packages* recipe in *Chapter 3, The Software Layer*, `meta-custom/recipes-example/helloworld/helloworld_1.0.bb`, as an example.

```
DESCRIPTION = "Simple helloworld application"
SECTION = "examples"
LICENSE = "MIT"
LIC_FILES_CHKSUM =
    "file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4
f302"

SRC_URI = "file://helloworld.c"

S = "${WORKDIR}"

do_compile() {
    ${CC} helloworld.c -o helloworld
}

do_install() {
    install -d ${D}${bindir}
    install -m 0755 helloworld ${D}${bindir}
}
```

Here, the `helloworld.c` source file is the following:

```
#include <stdio.h>

int main(void)
{
    return printf("Hello World");
}
```

The workflow steps are:

1. Start the package compilation from scratch.

```
$ cd /opt/yocto/fsl-community-bsp/
$ source setup-environment wandboard-quad
$ bitbake -c cleanall helloworld
```

This will erase the package's build folder, shared state cache, and downloaded package source.

2. Start a development shell:

```
$ bitbake -c devshell helloworld
```

This will fetch, unpack, and patch the `helloworld` sources and spawn a new shell with the environment ready for compilation. The new shell will change to the package's build directory.

3. Depending on the `SRC_URI` variable, the package's build directory can be revision controlled already. If not, as is the case in this example, we will create a local Git repository as follows:

```
$ git init
$ git add helloworld.c
$ git commit -s -m "Original revision"
```

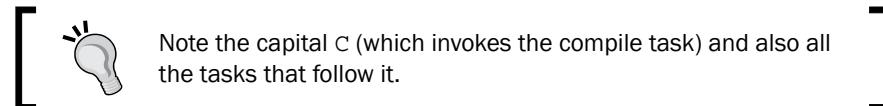
4. Perform the modifications we need; for example, change `helloworld.c` to print `Howdy world` as follows:

```
#include <stdio.h>

int main(void)
{
    return printf("Howdy World");
}
```

5. Exit devshell and build the package without erasing our modifications.

```
$ bitbake -C compile helloworld
```



6. Test your changes on the hardware by copying the generated package and installing it. Because you have only modified one package, the rest of the dependencies should be already installed in the running root filesystem. Run the following:

```
$ bitbake -e helloworld | grep ^WORKDIR=
WORKDIR="/opt/yocto/fsl-community-bsp/wandboard-
quad/tmp/work/cortexa9hf-vfp-neon-poky-linux-
gnueabi/helloworld/1.0-r0"
$ scp ${WORKDIR_PATH}/deploy-rpms/deploy-
rpms/cortexa9hf_vfp_neon/helloworld-1.0-
r0.cortexa9hf_vfp_neon.rpm root@<target_ip_address>:/
$ rpm -i /helloworld-1.0-r0.cortexa9hf_vfp_neon.rpm
```

This assumes the target's root filesystem has been built with the package-management feature and the helloworld package is added to the RM_WORK_EXCLUDE variable when using the rm_work class.

7. Go back to devshell and commit your change to the local Git repository as follows:

```
$ bitbake -c devshell helloworld
$ git add helloworld.c
$ git commit -s -m "Change greeting message"
```

8. Generate a patch into the recipe's patch directory:

```
$ git format-patch -1 -o /opt/yocto/fsl-community-
bsp/sources/meta-custom/recipes-
example/helloworld/helloworld-1.0
```

9. Finally, add the patch to the recipe's SRC_URI variable, as shown here:

```
SRC_URI = "file://helloworld.c \
           file://0001-Change-greeting-message.patch"
```

External source development

This workflow is recommended for development work once the application has been integrated into the Yocto build system. It can be used in conjunction with external development using an IDE, for example.

In the example recipe we saw earlier, the source file was placed on the `meta-custom` layer along with the metadata.

It is more common to have the recipe fetch directly from a revision control system like Git, so we will change the `meta-custom/recipes-example/helloworld/helloworld_1.0.bb` file to source from a Git directory as follows:

```
DESCRIPTION = "Simple helloworld application"
SECTION = "examples"
LICENSE = "MIT"
LIC_FILES_CHKSUM =
    "file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4
f302"

SRC_URI = "git://github.com/yoctocookbook/helloworld"

S = "${WORKDIR}/git"

do_compile() {
    ${CC} helloworld.c -o helloworld
}

do_install() {
    install -d ${D}${bindir}
    install -m 0755 helloworld ${D}${bindir}
}
```

We can then clone it into a local directory as follows:

```
$ cd /opt/yocto/
$ git clone git://github.com/yoctocookbook/helloworld
```

An alternative to using a remote revision controlled repository is to use a local one. To do so, follow these steps:

1. Create a local Git repository that will hold the source:

```
$ mkdir -p /opt/yocto/helloworld
$ cd /opt/yocto/helloworld
$ git init
```

2. Copy our `helloworld.c` file over here, and add it to the repository:

```
$ git add helloworld.c
```

3. Finally, commit it with a signature and a message:

```
$ git commit -s -m "Original revision"
```

In any case, we have the version-controlled source in a local directory. We will then configure our `conf/local.conf` file to work from it as follows:

```
INHERIT += "externalsrc"  
EXTERNALSRC_pn-helloworld = "/opt/yocto/helloworld"  
EXTERNALSRC_BUILD_pn-helloworld = "/opt/yocto/helloworld"
```

And build it with:

```
$ cd /opt/yocto/fsl-community-bsp/  
$ source setup-environment wandboard-quad  
$ bitbake helloworld
```

We can then work directly in the local folder without the risk of accidentally having BitBake erase our code. Once development is complete, the modifications to `conf/local.conf` are removed and the recipe will fetch the source from its original `SRC_URI` location.

Working with GNU make

GNU make is a make implementation for Linux systems. It is used by a wide variety of open source projects, including the Linux kernel. The build is managed by a `Makefile`, which tells make how to build the source code.

How to do it...

Yocto recipes inherit `base.bbclass` and hence their default behavior is to look for a `Makefile`, `makefile`, or `GNU Makefile` and use GNU make to build the package.

If your package already contains a `Makefile`, then all you need to worry about are the arguments that need to be passed to make. Make arguments can be passed using the `EXTRA_OEMAKE` variable, and a `do_install` override that calls the `oe_runmake install` needs to be provided, otherwise an empty install is run.

For example, the `logrotate` recipe is based on a `Makefile` and looks as follows:

```
SUMMARY = "Rotates, compresses, removes and mails system log  
files"  
SECTION = "console/utils"  
HOMEPAGE = "https://fedorahosted.org/logrotate/"
```

```
LICENSE = "GPLv2"

DEPENDS="coreutils poprt"

LIC_FILES_CHKSUM =
    "file://COPYING;md5=18810669f13b87348459e611d31ab760"

SRC_URI =
    "https://fedorahosted.org/releases/l/o/logrotate/logrotate-
    ${PV}.tar.gz \
SRC_URI [md5sum] = "99e08503ef24c3e2e3ff74cc5f3be213"
SRC_URI [sha256sum] =
    "f6ba691f40e30e640efa2752c1f9499a3f9738257660994de70a45fe00d12b64"

EXTRA_OEMAKE = ""

do_install(){
    oe_runmake install DESTDIR=${D} PREFIX=${D} MANDIR=${mandir}
    mkdir -p ${D}${sysconfdir}/logrotate.d
    mkdir -p ${D}${sysconfdir}/cron.daily
    mkdir -p ${D}${localstatedir}/lib
    install -p -m 644 examples/logrotate-default
    ${D}${sysconfdir}/logrotate.conf
    install -p -m 755 examples/logrotate.cron
    ${D}${sysconfdir}/cron.daily/logrotate
    touch ${D}${localstatedir}/lib/logrotate.status
}
```

See also

- ▶ For more information about GNU make, visit <https://www.gnu.org/software/make/manual/>

Working with the GNU build system

A `Makefile` is a good solution when you are always going to build and run your software on the same system, and things like `glibc` and `gcc` versions and the available library versions are known. However, most software need to be built and run in a variety of systems.

Getting ready

The GNU build system, or `autotools`, is a set of tools whose aim is to create a `Makefile` for your software in a variety of systems. It's made up of three main tools:

- ▶ `autoconf`: This parses the contents of a `configure.ac` file that describes the source code to be built and creates a `configure` script. This script will then be used to generate the final `Makefile`.
- ▶ `automake`: This parses the contents of a `Makefile.am` file and converts it into a `Makefile.in` file. This is then used by the `configure` script generated earlier to obtain a `config.status` script that gets automatically executed to obtain the final `Makefile`.
- ▶ `libtools`: This manages the creation of both static and dynamic libraries.

How to do it...

The Yocto build system contains classes with the required knowledge to build `autotools` packages. All your recipe needs to do is to inherit the `autotools` class and configure the arguments to be passed to the `configure` script in the `EXTRA_OECONF` variable. Usually, the `autotools` system understands how to install the software, so you do not need a `do_install` override.

There is a wide variety of open source projects that use `autotools` as the build system.

An example, `meta-custom/recipes-example/hello/hello_2.9.bb`, that does not need any extra `configure` options, follows:

```
DESCRIPTION = "GNU helloworld autotools recipe"
SECTION = "examples"

LICENSE = "GPLv3"
LIC_FILES_CHKSUM = "file://${COREBASE}/meta/files/common-licenses/GPL-3.0;md5=c79ff39f19dfec6d293b95dea7b07891"

SRC_URI = "${GNU_MIRROR}/hello/hello-${PV}.tar.gz"
SRC_URI[md5sum] = "67607d2616a0faaf5bc94c59dca7c3cb"
SRC_URI[sha256sum] =
"ecbb7a2214196c57ff9340aa71458e1559abd38f6d8d169666846935df191ea7"

inherit autotools gettext
```

See also

- ▶ For more information about the GNU build system, visit http://www.gnu.org/software/automake/manual/html_node/GNU-Build-System.html

Working with the CMake build system

The GNU make system is a great tool when you build exclusively for Linux systems. However, some packages are multiplatform and need a way to manage `Makefile` files on different operating systems. **CMake** is a cross-platform build system that can work not only with GNU make, but also Microsoft Visual Studio and Apple's Xcode.

Getting ready

The CMake tool parses the `CMakeLists.txt` files in every directory to control the build process. An example `CMakeLists.txt` file to compile the hello world example follows:

```
cmake_minimum_required(VERSION 2.8.10)
project(helloworld)
add_executable(helloworld helloworld.c)
install(TARGETS helloworld RUNTIME DESTINATION bin)
```

How to do it...

The Yocto build system also contains classes with the required knowledge to build CMake packages. All your recipe needs to do is to inherit the `cmake` class and configure the arguments to be passed to the `configure` script in the `EXTRA_OECMAKE` variable. Usually, the CMake system understands how to install the software, so you do not need a `do_install` override.

A recipe to build the `helloworld.C` example application, `meta-custom/recipes-example/helloworld-cmake/helloworld-cmake_1.0.bb`, follows:

```
DESCRIPTION = "Simple helloworld cmake application"
SECTION = "examples"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bf
cf8506ecda2f7b4
f302"
SRC_URI = "file://CMakeLists.txt \
           file://helloworld.c"
```

```
S = "${WORKDIR}"  
  
inherit cmake  
  
EXTRA_OECMAKE = ""
```

See also

- ▶ For more information about CMake, visit <http://www.cmake.org/documentation/>

Working with the SCons builder

SCons is also a multiplatform build system written in Python, with its configuration files also written in the same language. It also includes support for Microsoft Visual Studio among other features.

Getting ready

SCons parses the `SConstruct` files, and by default it does not propagate the environment into the build system. This is to avoid build issues caused by environment differences. This is a complication for Yocto, as it configures the environment with the cross-compilation toolchain settings.

SCons does not define a standard way to support cross-compilation, so every project will implement it differently. For a simple example as the hello world program, we can just initialize the `CC` and `PATH` variables from the external environment as follows:

```
import os  
env = Environment(CC = os.environ['CC'],  
                  ENV = {'PATH': os.environ['PATH']})  
env.Program("helloworld", "helloworld.c")
```

How to do it...

The Yocto build system also contains classes with the required knowledge to build SCons packages. All your recipe needs to do is to inherit the `SCons` class and configure the arguments to be passed to the configure script in the `EXTRA_OESCONS` variable. Although some packages using `SCons` might deal with installation through an `install` alias as required by the `SCons` class, your recipe will mostly need to provide a `do_install` task override.

An example recipe to build the `helloworld.c` example application, `meta-custom/recipes-example/helloworld-scons/helloworld-scons_1.0.bb`, follows:

```
DESCRIPTION = "Simple helloworld scons application"
SECTION = "examples"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bf8506ecda2f7b4f302"

SRC_URI = "file://SConstruct \
           file://helloworld.c"

S = "${WORKDIR}"

inherit scons

EXTRA_OESCONS = ""

do_install() {
    install -d ${D}/${bindir}
    install -m 0755 helloworld ${D}/${bindir}
}
```

See also

- ▶ For more information about SCons, visit <http://www.scons.org/doc/HTML/scons-user/>

Developing with libraries

Most applications make use of shared libraries, which saves system memory and disk space, as they are shared between different applications. Modularizing code into libraries also allows for easier versioning and code management.

This recipe will explain how to work with both static and shared libraries in Linux and Yocto.

Getting ready

By convention, library files start with the `lib` prefix.

There are basically two library types:

- ▶ **Static libraries** (.a): When the object code is linked and becomes part of the application
- ▶ **Dynamic libraries** (.so): Linked at compile time but not included in the application, so they need to be available at runtime. Multiple applications can share a dynamic library so they need less disk space.

Libraries are placed in the following standard root filesystem locations:

- ▶ /lib: Libraries required for startup
- ▶ /usr/lib: Most system libraries
- ▶ /usr/local/lib: Non-system libraries

Dynamic libraries follow certain naming conventions on running systems so that multiple versions can co-exist, so a library can be referenced by different names. Some of them are explained as follows:

- ▶ The linker name with the .so suffix; for example, libexample.so.
- ▶ The fully qualified name or soname, a symbolic link to the library name. For example, libexample.so.x, where x is the version number. Increasing the version number means the library is not compatible with previous versions.
- ▶ The real name. For example, libexample.so.x.y [.z], where x is the major version number, y is the minor version number, and the optional z is a release number. Increasing minor or release numbers retains compatibility.

In GNU glibc, starting an ELF binary calls a program loader, /lib/ld-linux-x. Here, x is the version number, which finds all the needed shared libraries. This process uses a couple of interesting files:

- ▶ /etc/ld.so.conf: This stores the directories searched by the loader
- ▶ /etc/ld.so.preload: This is used to override libraries

The ldconfig tool reads the ld.so.conf file and creates a cache file (/etc/ld.so.cache) to increase access speed.

The following environment variables can also be helpful:

- ▶ LD_LIBRARY_PATH: This is a colon-separated directory list to search libraries in. It is used when debugging or using non-standard library locations.
- ▶ LD_PRELOAD: This is used to override shared libraries.

Building a static library

We will build a static library, `libhelloworld`, from two source files, `hello.c` and `world.c`, and use it to build a hello world application. The source files for the library are presented here.

The following is the code for the `hello.c` file:

```
char * hello (void)
{
    return "Hello";
}
```

This is the code for `world.c` file:

```
char * world (void)
{
    return "World";
}
```

To build the library, follow these steps:

1. Configure the build environment:

```
$ source /opt/poky/1.7.1/environment-setup-cortexa9hf-vfp-
neon-poky-linux-gnueabi
```

2. Compile and link the library:

```
 ${CC} -c hello.c world.c
 ${AR} -cvq libhelloworld.a hello.o world.o
```

3. Verify the contents of the library:

```
 ${AR} -t libhelloworld.a
```

The application source code is presented next.

- ▶ For the `helloworld.c` file the following is the code:

```
#include <stdio.h>
int main (void)
{
    return printf("%s %s\n",hello(),world());
}
```

- ▶ To build it we run:

```
 ${CC} -o helloworld helloworld.c libhelloworld.a
```

- We can check which libraries it links with using `readelf`:

```
$ readelf -d helloworld
Dynamic section at offset 0x534 contains 24 entries:
  Tag          Type           Name/Value
  0x00000001  (NEEDED)      Shared library:
                           [libc.so.6]
```

Building a shared dynamic library

To build a dynamic library from the same sources, we would run:

```
 ${CC} -fPIC -g -c hello.c world.c
 ${CC} -shared -Wl,-soname,libhelloworld.so.1 -o libhelloworld.so.1.0
           hello.o world.o
```

We can then use it to build our `helloworld` C application, as follows:

```
 ${CC} helloworld.c libhelloworld.so.1.0 -o helloworld
```

And again, we can check the dynamic libraries using `readelf`, as follows:

```
$ readelf -d helloworld
Dynamic section at offset 0x6ec contains 25 entries:
  Tag          Type           Name/Value
  0x00000001  (NEEDED)      Shared library:
                           [libhelloworld.so.1]
  0x00000001  (NEEDED)      Shared library: [libc.so.6]
```

How to do it...

An example recipe for the static library example we just saw follows, `meta-custom/recipes-example/libhelloworld-static/libhelloworldstatic_1.0.bb`:

```
DESCRIPTION = "Simple helloworld example static library"
SECTION = "libs"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0b
                   cf8506ecdा2f7b4
                   f302"

SRC_URI = "file://hello.c \
           file://world.c \
           file://helloworld.pc"
```

```
S = "${WORKDIR}"  
  
do_compile() {  
    ${CC} -c hello.c world.c  
    ${AR} -cvq libhelloworld.a hello.o world.o  
}  
  
do_install() {  
    install -d ${D}${libdir}  
    install -m 0755 libhelloworld.a ${D}${libdir}  
}
```

By default, the configuration in `meta/conf/bitbake.conf` places all static libraries in a `-staticdev` package. It is also placed in the `sysroot` so that it can be used.

For a dynamic library, we would use the following recipe, `meta-custom/recipes-example/libhelloworld-dyn/libhelloworlddyn_1.0.bb`:

```
meta-custom/recipes-example/libhelloworld-dyn/libhelloworlddyn_1.0.bb  
DESCRIPTION = "Simple helloworld example dynamic library"  
SECTION = "libs"  
LICENSE = "MIT"  
LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0b  
cf8506ecd2f7b4  
f302"  
  
SRC_URI = "file://hello.c \  
          file://world.c \  
          file://helloworld.pc"  
  
S = "${WORKDIR}"  
  
do_compile() {  
    ${CC} -fPIC -g -c hello.c world.c  
    ${CC} -shared -Wl,-soname,libhelloworld.so.1 -o  
    libhelloworld.so.1.0 hello.o world.o  
}  
  
do_install() {  
    install -d ${D}${libdir}  
    install -m 0755 libhelloworld.so.1.0 ${D}${libdir}  
    ln -s libhelloworld.so.1.0  
    ${D}/${libdir}/libhelloworld.so.1  
    ln -s libhelloworld.so.1 ${D}/${libdir}/libhelloworld.so  
}
```

Usually we would list the library dependencies (if any) in the `RDEPENDS` variable, but this is not always needed as the build system performs some automatic dependency checking by inspecting both the library file and the `pkg-config` file and adding the dependencies it finds to `RDEPENDS` automatically.

Multiple versions of the same library can co-exist on the running system. For that, you need to provide different recipes with the same package name but different package revision. For example, we would have `libhelloworld-1.0.bb` and `libhelloworld-1.1.bb`.

And to build an application using the static library, we would create a recipe in `meta-custom/recipes-example/helloworld-static/helloworldstatic_1.0.bb`, as follows:

```
DESCRIPTION = "Simple helloworld example"
SECTION = "examples"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0b
cf8506ecda2f7b4
f302"

DEPENDS = "libhelloworld-static"

SRC_URI = "file://helloworld.c"

S = "${WORKDIR}"

do_compile() {
    ${CC} -o helloworld helloworld.c
    ${STAGING_LIBDIR}/libhelloworld.a
}

do_install() {
    install -d ${D}${bindir}
    install -m 0755 helloworld ${D}${bindir}
}
```

To build using the dynamic library, we would just need to change the recipe in `meta-custom/recipes-example/helloworld-shared/helloworldshared_1.0.bb` to `meta-custom/recipes-example/helloworld-shared/helloworldshared_1.0.bb`:

```
meta-custom/recipes-example/helloworld-shared/helloworldshared_1.0.bb
DESCRIPTION = "Simple helloworld example"
SECTION = "examples"
LICENSE = "MIT"
```

```
LIC_FILES_CHKSUM =
    "file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4
f302"

DEPENDS = "libhelloworld-dyn"

SRC_URI = "file://helloworld.c"

S = "${WORKDIR}"

do_compile() {
    ${CC} -o helloworld helloworld.c -lhelloworld
}

do_install() {
    install -d ${D}${bindir}
    install -m 0755 helloworld ${D}${bindir}
}
```

How it works...

Libraries should provide the information required to use them, such as `include` headers and library dependencies. The Yocto Project provides two ways for libraries to provide build settings:

- ▶ The `binconfig` class. This is a legacy class used for libraries that provide a `-config` script to provide build settings.
- ▶ The `pkgconfig` class. This is the recommended method for libraries to provide build settings.

A `pkg-config` build settings file has the `.pc` suffix, is distributed with the library, and is installed in a common location known to the `pkg-config` tool.

The `helloworld.pc` file for the dynamic library looks as follows:

```
prefix=/usr/local
exec_prefix=${prefix}
includedir=${prefix}/include
libdir=${exec_prefix}/lib

Name: helloworld
Description: The helloworld library
Version: 1.0.0
Cflags: -I${includedir}/helloworld
Libs: -L${libdir} -lhelloworld
```

However, for the static library, we would change the last line to:

```
Libs: -L${libdir} libhelloworld.a
```

A package wanting to use this .pc file would inherit the `pkgconfig` class.

There's more...

There's a provision for packages that build both a library and an executable but do not want both of them installed together. By inheriting the `lib_package` class, the package will create a separate `-bin` package with the executables.

See also

- ▶ More details regarding `pkg-config` can be found at <http://www.freedesktop.org/wiki/Software/pkg-config/>

Working with the Linux framebuffer

The Linux kernel provides an abstraction for the graphical hardware in the form of framebuffer devices. These allow applications to access the graphics hardware through a well-defined API. The framebuffer is also used to provide a graphical console to the Linux kernel, so that it can, for example, display colors and a logo.

In this recipe, we will explore how applications can use the Linux framebuffer to display graphics and video.

Getting ready

Some applications, especially in embedded devices, are able to access the framebuffer by mapping the memory and accessing it directly. For example, the `gstreamer` framework is able to work directly over the framebuffer, as is the Qt graphical framework.

Qt is a cross-platform application framework written in C++ and developed both by Digia, under the Qt company name, and the open source Qt project community.

For Qt applications, Poky provides a `qt4e-demo-image` and the FSL community BSP provides a `qte-in-use-image`, both of which include support for Qt4 Extended over the framebuffer. The provided framework also includes support for hardware acceleration – not only video but also 2D and 3D graphical acceleration provided through the OpenGL and OpenVG APIs.

How to do it...

To compile the Qt hello world application we saw in the *Developing Qt applications* recipe earlier, we could use the following `meta-custom/recipes-qt/qt-helloworld/qt-helloworld_1.0.bb` Yocto recipe:

```
DESCRIPTION = "Simple QT helloworld example"
SECTION = "examples"
LICENSE = "MIT"
LIC_FILES_CHKSUM =
    "file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4
     f302"

RDEPENDS_${PN} += "icu"

SRC_URI = "file://qt_hello_world.cpp \
           file://qt_hello_world.pro"

S = "${WORKDIR}"

inherit qt4e

do_install() {
    install -d ${D}${bindir}
    install -m 0755 qt_hello_world ${D}${bindir}
}
```

Here the `meta-custom/recipes-qt/qt-helloworld/qt-helloworld_1.0/qt_hello_world.cpp` source file is as follows:

```
#include <QApplication>
#include <QPushButton>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QPushButton hello("Hello world!");

    hello.show();
    return app.exec();
}
```

And the `meta-custom/recipes-qt/qt-helloworld/qt-helloworld-1.0/qt_hello_world.pro` project file is as follows:

```
SOURCES += \
    qt_hello_world.cpp
```

Then we add it to the image by using the following in your project's `conf/local.conf` file:

```
IMAGE_INSTALL_append = " qt-helloworld"
```

And we build the image with:

```
$ bitbake qt4e-demo-image
```

We can then program the SD card image, boot it, log in to the Wandboard, and launch the application by running:

```
# qt_hello_world -qws
```

The `-qws` command-line option is needed to run the server application.

How it works...

The framebuffer devices are located under `/dev`. The default framebuffer device is `/dev/fb0`, and if the graphics hardware provides more than one, they will be sequentially numbered.

By default, the Wandboard boots with two framebuffer devices, `fb0` and `fb1`. The first is the default video display, and the second one is an overlay plane that can be used to combine content on the display.

However, the i.MX6 SoC supports up to four displays, so it could have up to four framebuffer devices in addition to two overlay framebuffers.

You can change the default framebuffer used by applications with the `FRAMEBUFFER` environment variable. For example, if your hardware supports several framebuffers, you could use the second one by running:

```
# export FRAMEBUFFER=/dev/fb1
```

The framebuffer devices are memory mapped and you can perform file operations on them. For example, you can clear the contents of the screen by running:

```
# cat /dev/zero > /dev/fb0
```

Or copy it with:

```
# cat /dev/fb0 > fb.raw
```

You may even restore the contents with:

```
# cat fb.raw > /dev/fb0
```

User space programs can also interrogate the framebuffers or modify their configuration programmatically using `ioctl`s, or from the console by using the `fbset` application, which is included in Yocto's core images as a BusyBox applet.

```
# fbset -fb /dev/fb0
mode "1920x1080-60"
    # D: 148.500 MHz, H: 67.500 kHz, V: 60.000 Hz
    geometry 1920 1080 1920 1080 24
    timings 6734 148 88 36 4 44 5
    accel false
    rgba 8/16,8/8,8/0,0/0
endmode
```

You can configure the framebuffer HDMI device with a specific resolution, bits per pixel, and refresh rate by passing the `video` command-line option from the U-Boot bootloader to the Linux kernel. The specific format depends on the device framebuffer driver, and for the Wandboard it is as follows:

```
video=mxcfbn:dev=hDMI,<xres>x<yres>M[@rate]
```

Where:

- ▶ `n` is the framebuffer number
- ▶ `xres` is the horizontal resolution
- ▶ `yres` is the vertical resolution
- ▶ `M` specifies that the timings are to be calculated using the VESA coordinated video timings instead of from a look-up table
- ▶ `rate` is the refresh rate

For example, for the `fb0` framebuffer, you could use:

```
video=mxcfb0:dev=hDMI,1920x1080M@60
```



Note that after some time of inactivity, the virtual console will blank out.
To unblank the display, use:

```
# echo 0 > /sys/class/graphics/fb0/blank
```

There's more...

The FSL community BSP layer also provides a `fsl-image-multimedia` target image that includes the `gstreamer` framework, including plugins that make use of the hardware acceleration features within the i.MX6 SoC. A `fsl-image-multimedia-full` image is also provided, which extends the supported `gstreamer` plugins.

To build the `fsl-image-multimedia` image with framebuffer support, you need to remove the graphical distribution features by adding the following to your `conf/local.conf` file:

```
DISTRO_FEATURES_remove = "x11 directfb wayland"
```

And build the image with:

```
$ bitbake fsl-image-multimedia
```

The resulting `fsl-image-multimedia-wandboard-quad.sdcard` image at `tmp/deploy/images` can be programmed into a microSD card and booted.

The default Wandboard device tree defines an `mxcfb1` node as follows:

```
mxcfb1: fb@0 {
    compatible = "fsl,mxc_sdc_fb";
    disp_dev = "hdmi";
    interface_pix_fmt = "RGB24";
    mode_str ="1920x1080M@60";
    default_bpp = <24>;
    int_clk = <0>;
    late_init = <0>;
};
```

So, connecting a 1920x1080 HDMI monitor should show a virtual terminal with the Poky login prompt.

We can then use the `gstreamer` command-line tool, `gst-launch`, to construct `gstreamer` pipelines. For example, to view a hardware-accelerated video over the framebuffer, you can download the Big Bunny teaser full HD video file and play it over the framebuffer using the `gstreamer` framework's `gst-launch` command-line tool as follows:

```
# cd /home/root
# wget http://video.blender-testbuilds.de/download.blender.org/peach/trailer_1080p.mov
# gst-launch playbin2 uri=file:///home/root/trailer_1080p.mov
```

The video will use Freescale's h.264 video decoder plugin, `vpudec`, which makes use of the hardware video processing unit inside the i.MX6 SoC to decode the h.264 video.

You can see a list of the available i.MX6-specific plugins by running:

```
# gst-inspect | grep imx
h264.imx: mfw_h264decoder: h264 video decoder
audiopeq.imx: mfw_audio_pp: audio post equalizer
aiur.imx: webm: webm
aiur.imx: aiurdemux: aiur universal demuxer
mpeg2dec.imx: mfw_mpeg2decoder: mpeg2 video decoder
tvsrc.imx: tvsrc: v4l2 based tv src
ipucsc.imx: mfw_ipucsc: IPU-based video converter
mpeg4dec.imx: mfw_mpeg4aspdecoder: mpeg4 video decoder
vpu.imx: vpudec: VPU-based video decoder
vpu.imx: vpuenc: VPU-based video encoder
mp3enc.imx: mfw_mp3encoder: mp3 audio encoder
beep.imx: ac3: ac3
beep.imx: 3ca: ac3
beep.imx: beepdec: beep audio decoder
beep.imx: beepdec.vorbis: Vorbis decoder
beep.imx: beepdec.mp3: MP3 decoder
beep.imx: beepdec.aac: AAC LC decoder
isink.imx: mfw_isink: IPU-based video sink
v4lsink.imx: mfw_v4lsink: v4l2 video sink
v4lsrc.imx: mfw_v4lsrc: v4l2 based camera src
amrdec.imx: mfw_amrdecoder: amr audio decoder
```

See also

- ▶ The framebuffer API is documented in the Linux kernel documentation at <https://www.kernel.org/doc/Documentation/fb/api.txt>
- ▶ For more information regarding Qt for Embedded Linux, refer to <http://qt-project.org/doc/qt-4.8/qt-embedded-linux.html>
- ▶ Documentation for the gstreamer 0.10 framework can be found at <http://www.freedesktop.org/software/gstreamer-sdk/data/docs/2012.5/gstreamer-0.10/>

Using the X Windows system

The X Windows system provides the framework for a GUI environment – things like drawing and moving windows on the display and interacting with input devices like the mouse, the keyboard, and touchscreens. The protocol version has been X11 for over two decades, so it also known as X11.

Getting ready

The reference implementation for the X Windows system is the **X.Org** server, which is released under permissive licenses such as MIT. It uses a client/server model, with the server communicating with several client programs, serving user input, and accepting graphical output. The X11 protocol is network transparent so that the clients and the server may run on different machines, with different architectures and operating systems. However, mostly, they both run on the same machine and communicate using local sockets.

User interface specifications, such as buttons or menu styles, are not defined in X11, which leaves it to other window manager applications that are usually part of desktop environments, such as Gnome or KDE.

X11 has input and video drivers to handle the hardware. For example, it has a framebuffer driver, `fbdev`, that can output to a non-accelerated Linux framebuffer, and `evdev`, a generic Linux input device driver with support for mice, keyboards, tablets, and touchscreens.

The design of the X11 Windows systems makes it heavy for embedded devices, and although a powerful device like the quad-core i.MX6 has no trouble using it, many embedded devices choose other graphical alternatives. However, there are many graphical applications, mostly from the desktop environment, that run over the X11 Windows system.

The FSL community BSP layer provides a hardware-accelerated X video driver for the i.MX6 SoC, `xf86-video-imxfb-vivante`, which is included in the X11-based `core-image-sato` target image and other graphical images.

The X server is configured by an `/etc/X11/xorg.conf` file that configures the accelerated device as follows:

```
Section "Device"
    Identifier  "i.MX Accelerated Framebuffer Device"
    Driver      "vivante"
    Option      "fbdev"      "/dev/fb0"
    Option      "vivante_fbdev" "/dev/fb0"
    Option      "HWcursor"   "false"
EndSection
```

The graphical acceleration is provided by the Vivante GPUs included in the i.MX6 SoC.

Low-level X11 development is not recommended, and toolkits such as GTK+ and Qt are preferred. We will see how to integrate both types of graphical applications into our Yocto target image.

How to do it...

SATO is the default visual style for the Poky distribution based on **Gnome Mobile and Embedded (GMAE)**. It is a desktop environment based on GTK+ that uses the matchbox-window-manager. It has the peculiarity of showing one single fullscreen window at a time.

To build the GTK hello world application, `meta-custom/recipes-graphics/gtk-helloworld/gtk-helloworld-1.0/gtk_hello_world.c`, that we introduced earlier, as follows:

```
#include <gtk/gtk.h>

int main(int argc, char *argv[])
{
    GtkWidget *window;
    gtk_init (&argc, &argv);
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_widget_show (window);
    gtk_main ();
    return 0;
}
```

We can use the following `meta-custom/recipes-graphics/gtk-helloworld/gtk-helloworld_1.0.bb` recipe:

```
DESCRIPTION = "Simple GTK helloworld application"
SECTION = "examples"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bf8506ecda2f7b4f302"
SRC_URI = "file://gtk_hello_world.c"
S = "${WORKDIR}"
DEPENDS = "gtk+"
inherit pkgconfig
do_compile() {
```

```
 ${CC} gtk_hello_world.c -o helloworld `pkg-config --cflags --  
 libs gtk+-2.0`  
}  
  
do_install() {  
    install -d ${D}${bindir}  
    install -m 0755 helloworld ${D}${bindir}  
}
```

We can then add the package to the `core-image-sato` image by using:

```
IMAGE_INSTALL_append = " gtk-helloworld"
```

And we can build it, program it, and run the application from the serial terminal with:

```
# export DISPLAY=:0  
# helloworld
```

There's more...

Accelerated graphical output is also supported on the Qt framework, either directly on the framebuffer (like in the `qt4e-demo-image` target we saw before) or using the X11 server available in `core-image-sato`.

To build the Qt hello world source we introduced in the previous recipe but over X11, we can use the `meta-custom/recipes-qt/qtx11-helloworld/qtx11-helloworld_1.0.bb` Yocto recipe shown as follows::

```
DESCRIPTION = "Simple QT over X11 helloworld example"  
SECTION = "examples"  
LICENSE = "MIT"  
LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0b  
cf8506ecda2f7b4  
f302"  
  
RDEPENDS_${PN} += "icu"  
  
SRC_URI = "file://qt_hello_world.cpp \  
          file://qt_hello_world.pro"  
  
S = "${WORKDIR}"  
  
inherit qt4x11  
  
do_install() {
```

```
install -d ${D}${bindir}
install -m 0755 qt_hello_world ${D}${bindir}
}
```

We then need to add the Qt4 framework to the target image as well as the application.

```
EXTRA_IMAGE_FEATURES += "qt4-pkgs"
IMAGE_INSTALL_append = " qtx11-helloworld"
```

We can then build `core-image-sato` using the following command:

```
$ bitbake core-image-sato
```

Program and boot our target. Then run the application with:

```
# export DISPLAY=:0
# qt_hello_world
```

See also

- ▶ More information on the X.Org server can be found at <http://www.x.org>
- ▶ The Qt application framework documentation can be found at <https://qt-project.org/>
- ▶ More information and documentation about GTK+ can be found at <http://www.gtk.org/>

Using Wayland

Wayland is a display server protocol that is intended to replace the X Window system, and it is licensed under the MIT license.

This recipe will provide an overview of Wayland, including some key differences with the X Window system, and will show how to make use of it in Yocto.

Getting ready

The Wayland protocol follows a client/server model in which clients are the graphical applications requesting the display of pixel buffers on the screen, and the server, or compositor, is the service provider that controls the display of these buffers.

The Wayland compositor can be a Linux display server, an X application, or a special Wayland client. Weston is the reference Wayland compositor in the Wayland project. It is written in C and works with the Linux kernel APIs. It relies on evdev for the handling of input events.

Wayland uses **Direct Rendering Manager (DRM)** in the Linux kernel and does not need something like an X server. The client renders the window contents to a buffer shared with the compositor by itself, using a rendering library, or an engine like Qt or GTK+.

Wayland lacks the network transparency features of X, but it is likely that similar functionality will be added in the future.

It also has better security features than X and is designed to provide confidentiality and integrity. Wayland does not allow applications to look at the input of other programs, capture other input events, or generate fake input events. It also makes a better job out of protecting the Window outputs. However, this also means that it currently offers no way to provide some of the features we are used to in desktop X systems like screen capturing, or features common in accessibility programs.

Being lighter than X.Org and more secure, Wayland is better suited to use with embedded systems. If needed, X.Org can run as a client of Wayland for backwards compatibility.

However, Wayland is not as established as X11, and the Wayland-based images in Poky do not receive as much community attention as the X11-based ones.

How to do it...

Poky offers a `core-image-weston` image that includes the Weston compositor.

Modifying our GTK hello world example from the *Using the X Windows system* recipe to use GTK3 and run it with Weston is straightforward.

```
DESCRIPTION = "Simple GTK3 helloworld application"
SECTION = "examples"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0b
cf8506ecda2f7b4
f302"

SRC_URI = "file://gtk_hello_world.c"

S = "${WORKDIR}"

DEPENDS = "gtk+3"

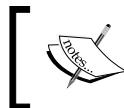
inherit pkgconfig

do_compile() {
    ${CC} gtk_hello_world.c -o helloworld `pkg-config --cflags --
libs gtk+-3.0`"
}
```

```
do_install() {
    install -d ${D}${bindir}
    install -m 0755 helloworld ${D}${bindir}
}
```

To build it, configure your `conf/local.conf` file by removing the `x11` distribution feature as follows:

```
DISTRO_FEATURES_remove = "x11"
```



You will need to build from scratch by removing both the `tmp` and `sstate-cache` directories when changing the `DISTRO_FEATURES` variable.



Add the application to the image with:

```
IMAGE_INSTALL_append = " gtk3-helloworld"
```

And build the image with:

```
$ cd /opt/yocto/fsl-community-bsp/
$ source setup-environment wandboard-quad
$ bitbake core-image-weston
```

Once the build finishes, you will find the microSD card image ready to be programmed under `tmp/deploy/images/wandboard-quad`.

You can then launch the application by running:

```
# export XDG_RUNTIME_DIR=/var/run/user/root
# helloworld
```

There's more...

The FSL community BSP release supports hardware-accelerated graphics in Wayland using the Vivante GPU included in the i.MX6 SoC.

This means that applications like `gstreamer` will be able to offer hardware-accelerated output when running with the Weston compositor.

Wayland support can also be found in graphical toolkits like Clutter and GTK3+.

See also

- ▶ You can find more information about Wayland on the project's web page at <http://wayland.freedesktop.org/>

Adding Python applications

In Yocto 1.7, Poky has support for building both Python 2 and Python 3 applications, and includes a small set of Python development tools in the `meta/recipes-devtools/python` directory.

A wider variety of Python applications are available in the `meta-python` layer included as part of `meta-openembedded`, which you can add to your `conf/bblayers.conf` file if you want to.

Getting ready

The standard tool for packaging Python modules is `distutils`, which is included for both Python 2 and Python 3. Poky includes the `distutils` class (`distutils3` in Python 3), which is used to build Python packages that use `distutils`. An example recipe in `meta-python` that uses the `distutils` class is `meta-python/recipes-devtools/python/python-pyusb_1.0.0a2.bb`.

```
SUMMARY = "PyUSB provides USB access on the Python language"
HOMEPAGE = "http://pyusb.sourceforge.net/"
SECTION = "devel/python"
LICENSE = "BSD"
LIC_FILES_CHKSUM =
    "file://LICENSE;md5=a53a9c39efcfb812e2464af14afab013"
DEPENDS = "libusb1"
PR = "r1"

SRC_URI = "\\\n    ${SOURCEFORGE_MIRROR}/pyusb/${SRCNAME}-${PV}.tar.gz \\\n"
SRC_URI[md5sum] = "9136b3dc019272c62a5b6d4eb624f89f"
SRC_URI[sha256sum] =
    "dacbf7d568c0bb09a974d56da66d165351f1ba3c4d5169ab5b734266623e1736"

SRCNAME = "pyusb"
S = "${WORKDIR}/${SRCNAME}-${PV}"

inherit distutils
```

However, `distutils` does not install package dependencies, allow package uninstallation, or allow us to install several versions of the same package, so it is only recommended for simple requirements. Hence, `setuptools` was developed to extend on `distutils`. It is not included in the standard Python libraries, but it is available in Poky. There is also a `setuptools` class in Poky (`setuptools3` for Python 3) that is used to build Python packages distributed with `setuptools`.

How to do it...

To build a Python hello world example application with `setuptools`, we would use a Yocto `meta-custom/recipes-python/python-helloworld/pythonhelloworld_1.0.bb` recipe as follows:

```
DESCRIPTION = "Simple Python setuptools hello world application"
SECTION = "examples"
LICENSE = "MIT"
LIC_FILES_CHKSUM =
    "file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4
f302"

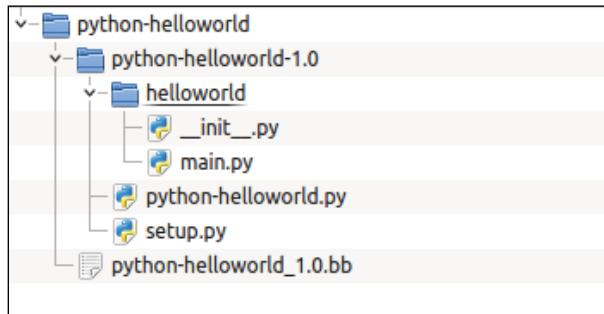
SRC_URI = "file://setup.py \
          file://python-helloworld.py \
          file://helloworld/__init__.py \
          file://helloworld/main.py"

S = "${WORKDIR}"

inherit setuptools

do_install_append () {
    install -d ${D}${bindir}
    install -m 0755 python-helloworld.py ${D}${bindir}
}
```

To create an example hello world package, we create the directory structure shown in the following screenshot:



Here is the code for the same directory structure:

```
$ mkdir -p meta-custom/recipes-python/python-helloworld/python-
  helloworld-1.0/helloworld/
$ touch meta-custom/recipes-python/python-helloworld/python-
  helloworld-1.0/helloworld/__init__.py
```

And create the following `meta-custom/recipes-python/python-helloworld/python-helloworld-1.0/setup.py` Python setup file:

```
import sys
from setuptools import setup

setup(
    name = "helloworld",
    version = "0.1",
    packages= ["helloworld"],
    author="Alex Gonzalez",
    author_email = "alex@example.com",
    description = "Hello World packaging example",
    license = "MIT",
    keywords= "example",
    url = "",
)
```

As well as the `meta-custom/recipes-python/python-helloworld/python-helloworld-1.0/helloworld/main.py` python file:

```
import sys

def main(argv=None):
    if argv is None:
```

```
    argv = sys.argv
    print "Hello world!"
    return 0
```

And a `meta-custom/recipes-python/python-helloworld/python-helloworld-1.0/python-helloworld.py` test script that makes use of the module:

```
#!/usr/bin/env python
import sys
import helloworld.main

if __name__ == '__main__':
    sys.exit(helloworld.main.main())
```

We can then add it to our image with:

```
IMAGE_INSTALL_append = " python-helloworld"
```

And build it using:

```
$ cd /opt/yocto/fsl-community-bsp/
$ source setup-environment wandboard-quad
$ bitbake core-image-minimal
```

Once programmed and booted, we can test the module by running the example script:

```
# /usr/bin/python-helloworld.py
Hello world!
```

There's more...

In `meta-python`, you can also find the `python-pip` recipe that will add the `pip` utility to your target image. It can be used to install packages from the **Python Package Index (PyPI)**.

You can add it to your image with:

```
IMAGE_INSTALL_append = " python-pip python-distribute"
```

You will need to add the `meta-openembedded/meta-python` layer to your `conf/bblayers.conf` file in order to build your image, and also the `python-distribute` dependency, which is needed by `python-pip`. Then you can build for the `core-image-minimal` image with:

```
$ cd /opt/yocto/fsl-community-bsp/
$ source setup-environment wandboard-quad
$ bitbake core-image-minimal
```

Once installed, you can use it from the target as follows:

```
# pip search <package_name>
# pip install <package_name>
```

Integrating the Oracle Java Runtime Environment

Oracle provides two specialized Java editions for embedded development:

- ▶ **Java SE embedded:** This is a large subset of the desktop version of the standard Java SE. It contains optimizations with respect to the standard edition, like size and memory usage, to adapt it to the needs of mid-sized embedded devices.
- ▶ **Java Micro Edition (ME):** This is targeted at headless low- and mid-range devices, and is a subset of Java SE complying with the **Connected Limited Device Configuration (CLDC)**, and including some extra features and tools for the embedded market. Oracle offers a couple of reference implementations, but Java ME will have to be individually integrated from source into specific platforms.

We will focus on Java SE embedded, which can be downloaded in binary format from the Oracle download site.

Java SE embedded is commercially licensed and requires royalty payments for embedded deployments.

Getting ready

Yocto has a `meta-oracle-java` layer that is meant to help in the integration of the official Oracle **Java Runtime Environment (JRE)** Version 7. However, installation without user intervention is not possible, as the Oracle's web page requires login and the acceptance of its license.

In Java SE embedded Version 7, Oracle offered both soft and hard floating point versions of headless and headful JREs for ARMv6/v7, and a headless version JRE for soft floating point user spaces for ARMv5. Java SE embedded version 7 provides two different **Java Virtual Machines (JVMs)** for ARM Linux:

- ▶ A client JVM optimized for responsiveness
- ▶ A server JVM identical to the client JVM but optimized for long-running applications

At the time of writing, the `meta-oracle-java` layer only has a recipe for the headless hard floating-point version with the client JVM. We will add recipes for the latest Java 7 SE embedded, which is update 75, for both headless and headful hard floating point JREs, which are appropriate to run on an i.MX6-based board like `wandboard-quad`.

How to do it...

To install the Java SE embedded runtime environment, first we need to clone the `meta-oracle-java` layer into our sources directory and add it to our `conf/bblayers.conf` file as follows:

```
$ cd /opt/yocto/fsl-community-bsp/sources  
$ git clone git://git.yoctoproject.org/meta-oracle-java
```

Then we need to explicitly accept the Oracle Java license by adding the following to our `conf/local.conf` file:

```
LICENSE_FLAGS_WHITELIST += "oracle_java"
```

We want to build the newest update available, so we add the following `meta-custom/recipes-devtools/oracle-java/oracle-jse-ejre-arm-vfpfhl-client-headless_1.7.0.bb` recipe to our `meta-custom` layer:

```
SUMMARY = "Oracle Java SE runtime environment binaries"  
  
JDK_JRE = "ejre"  
require recipes-devtools/oracle-java/oracle-jse.inc  
  
PV_UPDATE = "75"  
BUILD_NUMBER = "13"  
  
LIC_FILES_CHKSUM = "\  
    file://${WORKDIR}/${JDK_JRE}${PV}_${PV_UPDATE}/  
    COPYRIGHT;md5=0b204  
    bd2921accd6ef4a02f9c0001823 \  
    file://${WORKDIR}/${JDK_JRE}${PV}_${PV_UPDATE}/  
    THIRDPARTYLICENSES  
    ADME.txt;md5=f3a388961d24b8b72d412a079a878cdb \  
    "  
  
SRC_URI =  
    "http://download.oracle.com/otn/java/ejre/7u${PV_UPDATE}-  
    b${BUILD_NUMBER}/ejre-7u${PV_UPDATE}-fcs-b${BUILD_NUMBER}-linux-  
    arm-vfp-fhl-client_headless-18_dec_2014.tar.gz"
```

```
SRC_URI [md5sum] = "759ca6735d77778a573465b1e84b16ec"  
SRC_URI [sha256sum] =  
"ebb6499c62fc12e1471cff7431fec5407ace59477abd0f48347bf6e89c6bff3b"  
  
RPROVIDES_${PN} += "java2-runtime"
```

Try to build the recipe with the following:

```
$ bitbake oracle-jse-ejre-arm-vfp-hflt-client-headless
```

You will see that we get a checksum mismatch. This is caused by the license acceptance step in Oracle's website. To get around this, we will need to manually download the file into the `downloads` directory as specified in our project's `DL_DIR` configuration variable.

Then we can add the JRE to our target image:

```
IMAGE_INSTALL_append = " oracle-jse-ejre-arm-vfp-hflt-client-  
headless"
```

And build it with:

```
$ cd /opt/yocto/fsl-community-bsp/  
$ source setup-environment wandboard-quad  
$ bitbake core-image-minimal
```

We can now log in to the target and run it with:

```
# /usr/bin/java -version  
java version "1.7.0_75"  
Java(TM) SE Embedded Runtime Environment (build 1.7.0_75-b13,  
headless)  
Java HotSpot(TM) Embedded Client VM (build 24.75-b04, mixed mode)
```

We can also build the headful version using the following `meta-custom/recipes-devtools/oracle-java/oracle-jse-ejre-arm-vfphflt-client-headful_1.7.0.bb` recipe:

```
SUMMARY = "Oracle Java SE runtime environment binaries"  
  
JDK_JRE = "ejre"  
require recipes-devtools/oracle-java/oracle-jse.inc  
  
PV_UPDATE = "75"  
BUILD_NUMBER = "13"  
  
LIC_FILES_CHKSUM = "\
```

```
file://${WORKDIR}/${JDK_JRE}${PV}_${PV_UPDATE} /  
COPYRIGHT;md5=0b204  
bd2921accd6ef4a02f9c0001823 \  
file://${WORKDIR}/${JDK_JRE}${PV}_${PV_UPDATE} /  
THIRDPARTYLICENSE  
ADME.txt;md5=f3a388961d24b8b72d412a079a878cdb \  
"  
  
SRC_URI =  
"http://download.oracle.com/otn/java/ejre/7u${PV_UPDATE}-  
b${BUILD_NUMBER}/ejre-7u${PV_UPDATE}-fcs-b${BUILD_NUMBER}-linux-  
arm-vfp-hflt-client_headful-18_dec_2014.tar.gz"  
  
SRC_URI [md5sum] = "84dba4ffb47285b18e6382de2991edfc"  
SRC_URI [sha256sum] =  
"5738fffb8ce2582b6d7b39a3cbe16137d205961224899f8380eebe3922bae5c61"  
  
RPROVIDES_${PN} += "java2-runtime"
```

And add it to the target image with:

```
IMAGE_INSTALL_append = " oracle-jse-ejre-arm-vfp-hflt-client-  
headful"
```

And build core-image-sato with:

```
$ cd cd /opt/yocto/fsl-community-bsp/  
$ source setup-environment wandboard-quad  
$ bitbake core-image-sato
```

In this case, the reported Java version is:

```
# /usr/bin/java -version  
java version "1.7.0_75"  
Java(TM) SE Embedded Runtime Environment (build 1.7.0_75-b13)  
Java HotSpot(TM) Embedded Client VM (build 24.75-b04, mixed mode)
```

There's more...

The latest release at the time of this writing is Java SE embedded Version 8 update 33 (8u33).

Oracle offers the download of the JDK only, and a host tool, **jrecreate**, needs to be used to configure and create an appropriate JRE from the JDK. The tool allows us to choose between different JVMs (minimal, client, and server) as well as soft or hard floating point ABIs, extensions like JavaFX, locales, and several other tweakings to the JVM.

Oracle Java SE embedded Version 8 provides support for headful X11 development using Swing, AWT, and JavaFX only for ARMv7 hard floating point user spaces, and includes support for JavaFX (the graphical framework aimed to replace Swing and AWT) on the Freescale i.MX6 processor.

There is no Yocto recipe to integrate Java Version 8 at the time of this writing.

Integrating the Open Java Development Kit

The open source alternative to the Oracle Java SE embedded is the **Open Java Development Kit (OpenJDK)**, an open source implementation of Java SE licensed under the GPLv2, with the classpath exception, which means that applications are allowed to link without being bound by the GPL license.

This recipe will show how to build OpenJDK with Yocto and integrate the JRE into our target images.

Getting ready

The main components of OpenJDK are:

- ▶ The HotSpot Java Virtual Machine
- ▶ The **Java Class Library (JCL)**
- ▶ The Java compiler, **javac**

Initially, OpenJDK needed to be built using a proprietary JDK. However, the **IcedTea** project allowed us to build OpenJDK using the GNU classpath, the GNU compiler for Java (GCJ), and bootstrap a JDK to build OpenJDK. It also complements OpenJDK with some missing components available on Java SE like a web browser plugin and web start implementations.

Yocto can build meta-java using the `meta-jAVA` layer, which includes recipes for cross-compiling OpenJDK using IcedTea.

You can download OpenJDK from its Git repository at <http://git.yoctoproject.org/cgit/cgit.cgi/meta-jAVA/>.

Development discussions can be followed and contributed to by visiting the development mailing list at <http://lists.openembedded.org/mailman/listinfo/openembedded-devel>.

The `meta-jAVA` layer also includes recipes for a wide variety of Java libraries and VMs, and tools for application development like **ant** and **fastjar**.

How to do it...

To build OpenJDK 7, you need to clone the `meta-java` layer as follows:

```
$ cd /opt/yocto/fsl-community-bsp/sources/  
$ git clone http://git.yoctoproject.org/cgit.cgi/meta-java/
```

At the time of this writing, there is no 1.7 Dizzy branch yet, so we will work directly from the master branch.

Add the layer to your `conf/bblayers.conf` file:

```
+ ${BSPDIR}/sources/meta-java \  
"
```

And configure the project by adding the following to your `conf/local.conf` file:

```
PREFERRED_PROVIDER_virtual/java-initial = "cacao-initial"  
PREFERRED_PROVIDER_virtual/java-native = "jamvm-native"  
PREFERRED_PROVIDER_virtual/javac-native = "ecj-bootstrap-native"  
PREFERRED_VERSION_openjdk-7-jre = "25b30-2.3.12"  
PREFERRED_VERSION_icedtea7-native = "2.1.3"
```

You can then add the OpenJDK package to your image with:

```
IMAGE_INSTALL_append = " openjdk-7-jre"
```

And build the image of your choice:

```
$ cd /opt/yocto/fsl-community-bsp/  
$ source setup-environment wandboard-quad  
$ bitbake core-image-sato
```

When you run the target image, you will get the following Java version:

```
# java -version  
java version "1.7.0_25"  
OpenJDK Runtime Environment (IcedTea 2.3.12) (25b30-2.3.12)  
OpenJDK Zero VM (build 23.7-b01, mixed mode)
```

How it works...

To test the JVM, we can byte-compile a Java class on our host and copy it to the target to execute it. For instance, we can use the following simple `HelloWorld.java` example:

```
class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello World!");  
    }  
}
```

To byte-compile it in the host, we need to have a Java SDK installed. To install a Java SDK in Ubuntu, just run:

```
$ sudo apt-get install openjdk-7-jdk
```

To byte-compile the example, we execute:

```
$ javac HelloWorld.java
```

To run it, we copy the `HelloWorld.class` to the target, and from the same folder we run:

```
# java HelloWorld
```

There's more...

When using OpenJDK on a production system, it is recommended to always use the latest available release, which contains bug and security fixes. At the time of this writing, the latest OpenJDK 7 release is update 71 (jdk7u71b14), buildable with IcedTea 2.5.3, so the meta-java recipes should be updated.

See also

- ▶ Up-to-date information regarding openJDK can be obtained at <http://openjdk.java.net/>

Integrating Java applications

The meta-java layer also offers helper classes to ease the integration of Java libraries and applications into Yocto. In this recipe, we will see an example of building a Java library using the provided classes.

Getting ready

The meta-java layer provides two main classes to help with the integration of Java applications and libraries:

- ▶ **The Java bbclass:** This provides the default target directories and some auxiliary functions, namely:
 - oe_jarinstall: This installs and symlinks a JAR file
 - oe_makeclasspath: This generates a classpath string from JAR filenames
 - oe_java_simple_wrapper: This wraps your Java application in a shell script
- ▶ **The java-library bbclass:** This inherits the Java bbclass and extends it to create and install JAR files.

How to do it...

We will use the following meta-custom/recipes-java/java-helloworld/java-helloworld-1.0>HelloWorldSwing.java graphical Swing hello world as an example:

```
import javax.swing.JFrame;
import javax.swing.JLabel;

public class HelloWorldSwing {
    private static void createAndShowGUI() {
        JFrame frame = new JFrame("Hello World!");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JLabel label = new JLabel("Hello World!");
        frame.getContentPane().add(label);
        frame.pack();
        frame.setVisible(true);
    }
    public static void main(String[] args) {
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                createAndShowGUI();
            }
        });
    }
}
```

To integrate this `HelloWorldSwing` application, we can use a Yocto `meta-custom/recipes-java/java-helloworld/java-helloworld_1.0.bb` recipe as follows:

```
DESCRIPTION = "Simple Java Swing hello world application"
SECTION = "examples"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0b
cf8506ecda2f7b4
f302"

RDEPENDS_${PN} = "java2-runtime"

SRC_URI = "file://HelloWorldSwing.java"

S = "${WORKDIR}"

inherit java-library

do_compile() {
    mkdir -p build
    javac -d build `find . -name "*.java"`
    fastjar cf ${JARFILENAME} -C build .
}

BBCLASSEXTEND = "native"
```

The recipe is also buildable for the host native architecture. We can do this either by providing a separate `java-helloworld-native` recipe that inherits the `native` class or by using the `BBCLASSEXTEND` variable as we did earlier. In both cases, we could then use the `_class-native` and `_class-target` overrides to differentiate between native and target functionality.

Even though Java is byte-compiled and the compiled class will be the same for both, it still makes sense to add the native support explicitly.

How it works...

The `java-library` class will create a library package with the name `lib<package>-java`.

To add it to a target image, we would use:

```
IMAGE_INSTALL_append = " libjava-helloworld-java"
```

We can then decide whether we want to run the application with the Oracle JRE or OpenJDK. For OpenJDK, we will add the following packages to our image:

```
IMAGE_INSTALL_append = " openjdk-7-jre openjdk-7-common"
```

And for the Oracle JRE, we will use the following:

```
IMAGE_INSTALL_append = " oracle-jse-ejre-arm-vfp-hflt-client-
headful"
```

The available JREs do not currently run over the framebuffer or Wayland, so we will use an X11-based graphical image like `core-image-sato`:

```
$ cd /opt/yocto/fsl-community-bsp/
$ source setup-environment wandboard-quad
$ bitbake core-image-sato
```

We can then boot it, log in to the target, and execute the example with OpenJDK by running:

```
# export DISPLAY=:0
# java -cp /usr/share/java/java-helloworld.jar HelloWorldSwing
```

There's more...

At the time of this writing, OpenJDK as built from the `meta-java` layer master branch is not able to run X11 applications and will fail with this exception:

```
Exception in thread "main" java.awt.AWTError: Toolkit not found:
sun.awt.X11.XToolkit
    at java.awt.Toolkit$2.run(Toolkit.java:875)
    at java.security.AccessController.doPrivileged(Native
Method)
    at java.awt.Toolkit.getDefaultToolkit(Toolkit.java:860)
    at java.awt.Toolkit.getEventQueue(Toolkit.java:1730)
    at java.awt.EventQueue.invokeLater(EventQueue.java:1217)
    at javax.swing.SwingUtilities.invokeLater(SwingUtilities.
java:1287)
    at HelloWorldSwing.main(HelloWorldSwing.java:17)
```

However, the precompiled Oracle JRE runs the application without issues with:

```
# export DISPLAY=:0
# /usr/bin/java -cp /usr/share/java/java-helloworld.jar
HelloWorldSwing
```

If you see build errors when building packages with the Oracle JRE, try using a different package format, for example, IPK, by adding the following to your conf/local.conf configuration file:

```
PACKAGE_CLASSES = "package_ipk"
```

This is due to dependency problems in the meta-oracle-java layer with the RPM package manager, as explained in the layer's README file.



5

Debugging, Tracing, and Profiling

In this chapter, we will cover the following recipes:

- ▶ Analyzing core dumps
- ▶ Native GDB debugging
- ▶ Cross GDB debugging
- ▶ Using strace for application debugging
- ▶ Using the kernel's performance counters
- ▶ Using static kernel tracing
- ▶ Using dynamic kernel tracing
- ▶ Using dynamic kernel events
- ▶ Exploring Yocto's tracing and profiling tools
- ▶ Tracing and profiling with perf
- ▶ Using SystemTap
- ▶ Using OProfile
- ▶ Using LTTng
- ▶ Using blktrace

Introduction

Debugging an embedded Linux product is a common task not only during development, but also in deployed production systems.

Application debugging in embedded Linux is different from debugging in a traditional embedded device in that we don't have a flat memory model with an operating system and applications sharing the same address space. Instead, we have a virtual memory model with the Linux operating system, sharing the address space and assigning virtual memory areas to running processes.

With this model, the mechanisms used for kernel and user space debugging differ. For example, the traditional model of using a JTAG-based hardware debugger is useful for kernel debugging, but unless it knows about the user space processes memory mapping, it will not be able to debug user space applications.

Application debugging is approached with the use of a user space debugger service. We have seen an example of this methodology in action with the TCF agent used in the Eclipse GDB. The other commonly used agent is the **gdbserver**, which we will use in this chapter.

Finally we will explore the area of tracing and profiling. Tracing is a low-level logging of frequent system events, and the statistical analysis of these captured traces is called profiling.

We will use some of the tools embedded Linux and Yocto offer to trace and profile our systems so that they run to their maximum potential.

Analyzing core dumps

Even after extensive quality assurance testing, embedded systems in-field also fail and need to be debugged. Moreover, often the failure is not something that can be easily reproduced in a laboratory environment, so we are left with production, often hardened system, to debug.

Assuming we have designed our system with the aforementioned scenario in mind, our first debugging choice is usually to extract as much information about the failing system—for example, by obtaining and analyzing a core dump of the misbehaving processes.

Getting ready

In the process of debugging embedded Linux systems, we can use the same toolbox as standard Linux systems. One of the tools enables applications to generate into the disk a memory core dump upon crashing. This assumes that we have enough disk space to store the application's entire memory map, and that writing to disk is quick enough that it will not drag the system to a halt.

Once the memory core dump is generated, we use the host's GDB to analyze the core dump. GDB needs to have debug information available. Debug information can be in the executable itself—for example, when we install the `-dbg` version of a package, or we configure our project to not strip binaries—or can be kept in a separate file. To install debug information separately from the executable, we use the `dbg-pkgs` feature. By default, this installs the debug information of a package in a `.debug` directory in the same location as the executable itself. To add debug information for all packages in a target image, we add the following to our `conf/local.conf` configuration file:

```
EXTRA_IMAGE_FEATURES += "dbg-pkgs"
```

We can then build an appropriate toolchain generated to match our filesystem, as we saw in the *Preparing and using an SDK* recipe in Chapter 4, Application Development. The core dump contains build IDs for the executables and libraries in use at the time of the crash, so it's important to match the toolchain and the target image.

How to do it...

We can display the limits of the system-wide resources with the `ulimit` tool. We are interested in the core file size, which by default is set to zero to avoid the creation of application core dumps. In our failing system, preferably in a test environment, make your application dump a memory core upon crashing with:

```
$ ulimit -c unlimited
```

You can then verify the change with:

```
$ ulimit -a
-f: file size (blocks)          unlimited
-t: cpu time (seconds)         unlimited
-d: data seg size (kb)         unlimited
-s: stack size (kb)            8192
-c: core file size (blocks)    unlimited
-m: resident set size (kb)    unlimited
-l: locked memory (kb)        64
-p: processes                  5489
-n: file descriptors           1024
-v: address space (kb)         unlimited
-w: locks                      unlimited
-e: scheduling priority         0
-r: real-time priority          0
```

For this example, we will be using the `wvdial` application in a real segmentation fault scenario. The purpose is not to debug the application itself but to showcase the methodology used for core dump analysis; so, details regarding the application-specific configuration and system setup are not provided. However, being a real crash, the example is more illustrative.

To run `wvdial` on the target, use the following code:

```
# wvdial
--> WvDial: Internet dialer version 1.61
--> Initializing modem.
--> Sending: ATZ
ATZ
OK
--> Sending: ATQ0 V1 E1 S0=0 &C1 &D2 +FCLASS=0
ATQ0 V1 E1 S0=0 &C1 &D2 +FCLASS=0
OK
--> Sending: AT+CGDCONT=1,"IP","internet"
AT+CGDCONT=1,"IP","internet"
OK
--> Modem initialized.
--> Idle Seconds = 3000, disabling automatic reconnect.
Segmentation fault (core dumped)
```

The application will create a core file in the same folder, which you can then copy to your host system to analyze.



You can also simulate a core dump by sending a `SIGQUIT` signal to a running process. For example, you could force the `sleep` command to core dump with a `SIGQUIT` signal as follows:

```
$ ulimit -c unlimited
$ sleep 30 &
$ kill -QUIT <sleep-pid>
```

How it works...

Once in possession of the core dump, use the cross GDB in the host to load it and get some useful information, such as the backtrace, using the following steps:

1. First set up the environment in the host:

```
$ cd /opt/poky/1.7.1/
$ source environment-setup-cortexa9hf-vfp-neon-poky-linux-
gnueabi
```

2. You can then start the cross GDB debugger, passing it a debug version of the application. Debug versions are stored in the `sysroot` file in the same location as the unstripped binary, but under a `.debug` directory.

The whole GDB banner is showed below but will be omitted in future examples.

```
$ arm-poky-linux-gnueabi-gdb /opt/yocto/fsl-community-  
bsp/wandboard-quad/tmp/work/cortexa9hf-vfp-neon-poky-linux-  
gnueabi/wvdial/1.61-r0/packages-split/wvdial-  
dbg/usr/bin/.debug/wvdial core  
GNU gdb (GDB) 7.7.1  
Copyright (C) 2014 Free Software Foundation, Inc.  
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>  
This is free software: you are free to change and redistribute  
it.  
There is NO WARRANTY, to the extent permitted by law. Type  
"show copying"  
and "show warranty" for details.  
This GDB was configured as "--host=x86_64-pokysdk-linux --  
target=arm-poky-linux-gnueabi".  
Type "show configuration" for configuration details.  
For bug reporting instructions, please see:  
<http://www.gnu.org/software/gdb/bugs/>.  
Find the GDB manual and other documentation resources online  
at:  
<http://www.gnu.org/software/gdb/documentation/>.  
For help, type "help".  
Type "apropos word" to search for commands related to  
"word"....  
Reading symbols from /opt/yocto/fsl-community-bsp/wandboard-  
quad/tmp/work/cortexa9hf-vfp-neon-poky-linux-  
gnueabi/wvdial/1.61-r0/packages-split/wvdial-  
dbg/usr/bin/.debug/wvdial...done.  
[New LWP 1050]  
  
warning: Could not load shared library symbols for 14  
libraries, e.g. /usr/lib/libwvstreams.so.4.6.  
Use the "info sharedlibrary" command to see the complete  
listing.  
Do you need "set solib-search-path" or "set sysroot"?  
Core was generated by `wvdial'.
```

```
Program terminated with signal SIGSEGV, Segmentation fault.  
#0 0x76d524c4 in ?? ()
```

3. Now point GDB to the location of the toolchain's sysroot:

```
(gdb) set sysroot /opt/poky/1.7.1/sysroots/cortexa9hf-vfp-  
neon-poky-linux-gnueabi/  
Reading symbols from /opt/poky/1.7.1/sysroots/cortexa9hf-vfp-  
neon-poky-linux-gnueabi/usr/lib/libwvstreams.so.4.6...Reading  
symbols from  
/opt/poky/1.7.1/sysroots/cortexa9hf-vfp-neon-poky-linux-  
gnueabi/usr/lib/.debug/libwvstreams.so.4.6...done.  
done.  
Loaded symbols for /opt/poky/1.7.1/sysroots/cortexa9hf-vfp-  
neon-poky-linux-gnueabi/usr/lib/libwvstreams.so.4.6  
Reading symbols from /opt/poky/1.7.1/sysroots/cortexa9hf-vfp-  
neon-poky-linux-gnueabi/usr/lib/libwvutils.so.4.6...Reading  
symbols from /opt/poky/1.7.1/sysroots/cortexa9hf-vfp-neon-  
poky-linux-gnueabi/usr/lib/.debug/libwvutils.so.4.6...done.  
done.  
[...]  
Loaded symbols for /opt/poky/1.7.1/sysroots/cortexa9hf-vfp-  
neon-poky-linux-gnueabi/lib/libdl.so.2
```

4. You can now inquire GDB for the application's backtrace as follows:

```
(gdb) bt  
#0 0x76d524c4 in WvTaskMan::_stackmaster () at  
utils/wvtask.cc:416  
#1 0x00000000 in ?? ()
```

See also

- ▶ The usage documentation for GDB found at <http://www.gnu.org/software/gdb/documentation/>

Native GDB debugging

On devices as powerful as the Wandboard, native debugging is also an option to debug sporadic failures. This recipe will explore the native debugging method.

Getting ready

For native development and debugging, Yocto offers the `-dev` and `-sdk` target images. To add developing tools to the `-dev` images, we can use the `tools-sdk` feature. We also want to install debug information and debug tools, and we do this by adding the `dbg-pkgs` and `tools-debug` features to our image. For example, for `core-image-minimal-dev`, we would add the following to our `conf/local.conf` file:

```
EXTRA_IMAGE_FEATURES += "tools-sdk dbg-pkgs tools-debug"
```

To prepare a development-ready version of the `core-image-minimal-dev` target image, we would execute the following commands:

```
$ cd /opt/yocto/fsl-community-bsp/  
$ source setup-environment wandboard-quad  
$ bitbake core-image-minimal-dev
```

We will then program the development image to our target.

How to do it...

Once the target has booted, you can start the `wvdial` application through the native GDB using the following steps:

1. In the target command prompt, start the GDB debugger with the application as argument:

```
$ gdb wvdial
```

2. Now instruct GDB to run the application:

```
(gdb) run  
Starting program: /usr/bin/wvdial  
Cannot access memory at address 0x0  
Cannot access memory at address 0x0  
  
Program received signal SIGILL, Illegal instruction.  
0x7698afe8 in ?? () from /lib/libcrypto.so.1.0.0  
(gdb) sharedlibrary libcrypto  
Symbols already loaded for /lib/libcrypto.so.1.0.0
```

3. Then request to print a backtrace:

```
(gdb) bt
#0 0x7698afe8 in ?? () from /lib/libcrypto.so.1.0.0
#1 0x769878e8 in OPENSSL_cpuid_setup () from /lib/libcrypto.
so.1.0.0
#2 0x76fe715c in ?? () from /lib/ld-linux-armhf.so.3
Cannot access memory at address 0x48535540
```

This is not the same backtrace you got when analyzing the core dump. What is going on here? The clue is on libcrypto, part of the OpenSSL library. OpenSSL probes the capabilities of the system by trying each capability and trapping the illegal instruction errors. So the SIGILL signal you are seeing during startup is normal and you should instruct GDB to continue.

4. Instruct GDB to continue:

```
(gdb) c
Continuing.
--> WvDial: Internet dialer version 1.61
--> Initializing modem.
--> Sending: ATZ
ATZ
OK
--> Sending: ATQ0 V1 E1 S0=0 &C1 &D2 +FCLASS=0
ATQ0 V1 E1 S0=0 &C1 &D2 +FCLASS=0
OK
--> Sending: AT+CGDCONT=1,"IP","internet"
AT+CGDCONT=1,"IP","internet"
OK
--> Modem initialized.
--> Idle Seconds = 3000, disabling automatic reconnect.

Program received signal SIGSEGV, Segmentation fault.
0x76db74c4 in WvTaskMan::_stackmaster() () from /usr/lib/
libwvbase.so.4.6
```

This result is now compatible with the core dump you saw in the previous recipe.

There's more...

When debugging applications, it is sometimes useful to reduce the level of optimization used by the compiler. This will reduce the application's performance but will facilitate debugging by improving the accuracy of the debug information. You can configure the build system to reduce optimization and add debug information by adding the following line of code to your `conf/local.conf` file:

```
DEBUG_BUILD = "1"
```

By using this configuration, the optimization is reduced from `FULL_OPTIMIZATION` (`-O2`) to `DEBUG_OPTIMIZATION` (`-O -fno-omit-frame-pointer`). But sometimes this is not enough, and you may like to build with no optimization. You can achieve this by overriding the `DEBUG_OPTIMIZATION` variable either globally or for a specific recipe.

See also

- ▶ The example on using a debug-optimized build in the upcoming recipe on *Cross GDB debugging*

Cross GDB debugging

When we run a cross compiled GDB in the host, which connects to a native gdbserver running on the target, it is referred to as cross debugging. This is the same scenario we saw in the *Using the Eclipse IDE* recipe earlier, except that Eclipse uses the **Target Communications Framework (TCF)**. Cross debugging has the advantage of not needing debug information on target images, as they are already available in the host.

This recipe will show how to use a cross GDB and gdbserver.

Getting ready

To include gdbserver in your target image, you can use an `-sdk` image, or you can add the `tools-debug` feature to your image by adding the following to your `conf/local.conf` configuration file:

```
EXTRA_IMAGE_FEATURES += "tools-debug"
```

So that GDB can access debug information of the shared libraries and executables, add the following to the `conf/local.conf` file:

```
EXTRA_IMAGE_FEATURES += "dbg-pkgs"
```

The images running on the target and the toolchain's sysroot need to match. For example, if you are using core-image-minimal images, the toolchain needs to have been generated in the same project with:

```
$ bitbake -c populate_sdk core-image-minimal
```

This will generate a sysroot containing debug information for binaries and libraries.

How to do it...

Once the toolchain is installed, you can run the application to be debugged on the target using gdbserver—in this case, wvdial—in the following steps:

1. Launch gdbserver with the application to run as argument:

```
# gdbserver localhost:1234 /usr/bin/wvdial
Process wvdial created; pid = 879
Listening on port 1234
```

The gdbserver is launched listening on localhost on a random 1234 port and is waiting for a connection from the remote GDB.

2. In the host, you can now set up the environment using the recently installed toolchain:

```
$ cd /opt/poky/1.7.1/
$ source environment-setup-cortexa9hf-vfp-neon-poky-linux-
gnueabi
```

You can then launch the cross GDB, passing to it the absolute path to the debug version of the application to debug, which is located in a .debug directory on the sysroot:

```
$ arm-poky-linux-gnueabi-gdb
/opt/poky/1.7.1/sysroots/cortexa9hf-vfp-neon-poky-linux-
gnueabi/usr/bin/.debug/wvdial
Reading symbols from /opt/poky/1.7.1/sysroots/cortexa9hf-vfp-
neon-poky-linux-gnueabi/usr/bin/.debug/wvdial...done.
(gdb)
```

3. Next configure GDB to consider all files as trusted so that it auto loads whatever it needs:

```
(gdb) set auto-load safe-path /
```

4. Also as you know, wvdial will generate a SIGILL signal that will interrupt our debugging session, instruct GDB not to stop when that signal is seen:

```
(gdb) handle SIGILL nostop
```

5. You can then connect to the remote target on the 1234 port with:

```
(gdb) target remote <target_ip>:1234
Remote debugging using 192.168.128.6:1234
Cannot access memory at address 0x0
0x76fd7b00 in ?? ()
```

6. The first thing to do is to set sysroot so that GDB is able to find dynamically loaded libraries:

```
(gdb) set sysroot /opt/poky/1.7.1/sysroots/cortexa9hf-vfp-
neon-poky-linux-gnueabi
Reading symbols from /opt/poky/1.7.1/sysroots/cortexa9hf-
vfp-neon-poky-linux-gnueabi/lib/ld-linux-
armhf.so.3...done.
Loaded symbols for /opt/poky/1.7.1/sysroots/cortexa9hf-vfp-
neon-poky-linux-gnueabi/lib/ld-linux-armhf.so.3
```

7. Type c to continue with the program's execution. You will see wvdial continuing on the target:

```
--> WvDial: Internet dialer version 1.61
--> Initializing modem.
--> Sending: ATZ
ATZ
OK
--> Sending: ATQ0 V1 E1 S0=0 &C1 &D2 +FCLASS=0
ATQ0 V1 E1 S0=0 &C1 &D2 +FCLASS=0
OK
--> Sending: AT+CGDCONT=1,"IP","internet"
AT+CGDCONT=1,"IP","internet"
OK
--> Modem initialized.
--> Idle Seconds = 3000, disabling automatic reconnect.
```

8. You will then see GDB intercepting a SIGILL and SEGSEGV signal on the host:

```
Program received signal SIGILL, Illegal instruction.
```

```
Program received signal SIGSEGV, Segmentation fault.  
0x76dc14c4 in WvTaskMan::_stackmaster () at  
    utils/wvtask.cc:416  
416      utils/wvtask.cc: No such file or directory.
```

9. You can now ask to see a backtrace:

```
(gdb) bt  
#0 0x76dc14c4 in WvTaskMan::_stackmaster () at  
    utils/wvtask.cc:416  
#1 0x00000000 in ?? ()
```

Although limited, this backtrace could still be useful to debug the application.

How it works...

We see a limited backtrace because the compiled binaries are not suitable for debugging, as they omit stack frames. To keep information on stack frames, add the following to the `conf/local.conf` configuration file:

```
DEBUG_BUILD = "1"
```

This changes the compilation flags to debug optimization as follows:

```
DEBUG_OPTIMIZATION = "-O -fno-omit-frame-pointer ${DEBUG_FLAGS} -pipe"
```

The `-fno-omit-frame-pointer` flag will tell `gcc` to keep stack frames. The compiler will also reduce the optimization level to provide a better debugging experience.

A debug build will also make it possible to trace variables and set breakpoints and watchpoints, as well as other common debugging features.

After building and installing the target images and toolchain again, you can now follow the same process as in the preceding recipe:

1. Use the following code for connecting to the remote target:

```
(gdb) target remote <target_ip>:1234  
Remote debugging using 192.168.128.6:1234  
warning: Unable to find dynamic linker breakpoint function.  
GDB will be unable to debug shared library initializers  
and track explicitly loaded dynamic code.  
Cannot access memory at address 0x0  
0x76fdd800 in ?? ()
```

Set the sysroot as follows:

```
(gdb) set sysroot /opt/poky/1.7.1/sysroots/cortexa9hf-vfp-
neon-poky-linux-gnueabi
Reading symbols from /opt/poky/1.7.1/sysroots/cortexa9hf-
vfp-neon-poky-linux-gnueabi/lib/ld-linux-
armhf.so.3...done.
Loaded symbols for /opt/poky/1.7.1/sysroots/cortexa9hf-vfp-
neon-poky-linux-gnueabi/lib/ld-linux-armhf.so.3
```

2. Once you are done with the setup, instruct the program to continue as follows:

```
(gdb) c
Continuing.
```

```
Program received signal SIGILL, Illegal instruction.
```

```
Program received signal SIGABRT, Aborted.
0x76b28bb4 in __GI_raise (sig=sig@entry=6) at
./sysdeps/unix/sysv/linux/raise.c:55
55      .../sysdeps/unix/sysv/linux/raise.c: No such file or
directory.
(gdb) bt
#0 0x76b28bb4 in __GI_raise (sig=sig@entry=6) at
./sysdeps/unix/sysv/linux/raise.c:55
#1 0x76b2cabc in __GI_abort () at abort.c:89
#2 0x76decfa8 in __assert_fail (__assertion=0x76df4600
"magic_number == -0x123678",
__file=0x1 <error: Cannot access memory at address
0x1>, __line=427,
__function=0x76df4584
<WvTaskMan:::_stackmaster()::__PRETTY_FUNCTION__> "static
void WvTaskMan:::_stackmaster()"
    at utils/wvcrashbase.cc:98
#3 0x76dc58c8 in WvTaskMan::_stackmaster () at
utils/wvtask.cc:427
Cannot access memory at address 0x123678
#4 0x00033690 in ?? ()
Cannot access memory at address 0x123678
Backtrace stopped: previous frame identical to this frame
(corrupt stack?)
```

You can now see a complete backtrace.

Using strace for application debugging

Debugging does not always involve working with source code. Sometimes it is a change in an external factor that is causing the problem.

Strace is a tool that is useful for scenarios where we are looking for problems outside of the binary itself; for example configuration files, input data, and kernel interfaces. This recipe will explain how to use it.

Getting ready

To include strace in your system, add the following to your `conf/local.conf` file:

IMAGE_INSTALL_append = " strace"

Strace is also part of the tools-debug image feature, so you can also add it with:

EXTRA IMAGE FEATURES += "tools-debug"

Strace is also included in the -sdk images.

Before starting, we will also include `pgrep`, a process utility that will make our debugging easier by looking up process IDs by name. To do so, add the following to your `conf/local.conf` configuration file:

```
IMAGE_INSTALL_append = " procps"
```

How to do it...

When printing a system call, strace prints the values passed to the kernel or returned from the kernel. The verbose option prints more details for some system calls.

For example, filtering just the `sendto()` system calls from a single ping looks as follows:

How it works...

Strace allows the monitoring of system calls of running processes into the Linux kernel. It uses the `ptrace()` system call to do so. This means that other programs that use `ptrace()`, such as `gdb`, will not run simultaneously.

Strace is a disruptive monitoring tool, and the process being monitored will slow down and create many more context switches. A generic way of running strace on a given program is:

```
strace -f -e <filter> -t -s<num> -o <log file>.strace <program>
```

The arguments are explained below:

- ▶ **f**: Tells strace to trace all child processes.
- ▶ **e**: Filters the output to a selection of comma separated system calls.
- ▶ **t**: Prints absolute timestamps. Use **r** for timestamps relative to the last syscall, and **T** to add the time spent in the syscall.
- ▶ **s**: Increases the maximum length of strings from the default of 32.
- ▶ **o**: Redirects the output to a file that can then be analyzed offline.

It can also attach to running processes using the following command:

```
$ strace -p $( pgrep <program> )
```

Or several instances of a process using the following command:

```
$ strace $( pgrep <program> | sed 's/^/-p' )
```

To detach, just press *Ctrl + C*.

See also

- ▶ The corresponding man pages for more information about strace at <http://man7.org/linux/man-pages/man1/strace.1.html>

Using the kernel's performance counters

Hardware performance counters are perfect for code optimization, especially in embedded systems with a single workload. They are actively used by a wide range of tracing and profiling tools. This recipe will introduce the Linux performance counters subsystem and show how to use it.

Getting ready

The **Linux Kernel Performance Counters Subsystem (LPC)**, commonly known as `linux_perf`, is an abstraction interface to different CPU-specific performance measurements. The `perf_events` subsystem not only exposes hardware performance counters from the CPU, but also kernel software events using the same API. It also allows the mapping of events to processes, although this has a performance overhead. Further, it provides generalized events which are common across architectures.

Events can be categorized into three main groups:

- ▶ **Software events:** Based on kernel counters, these events are used for things such as context switches and minor faults tracking.
- ▶ **Hardware events:** These come from the processor's CPU **Performance Monitoring Unit (PMU)** and are used to track architecture-specific items, such as the number of cycles, cache misses, and so on. They vary with each processor type.
- ▶ **Hardware cache events:** These are common hardware events that will only be available if they actually map to a CPU hardware event.

To know whether `perf_event` support is available for your platform, you can check for the existence of the `/proc/sys/kernel/perf_event_paranoid` file. This file is also used to restrict access to the performance counters, which by default are set to allow both user and kernel measurement. It can have the following values:

- ▶ 2: Only allows user-space measurements
- ▶ 1: Allows both kernel and user measurements (default)
- ▶ 0: Allows access to CPU-specific data but not raw tracepoint samples
- ▶ -1: No restrictions

The i.MX6 SoC has a Cortex-A9 CPU which includes a PMU, providing six counters to gather statistics on the operation of the processor and memory, each one of them able to monitor any of 58 available events.

You can find a description of the available events in the *Cortex-A9 Technical Reference Manual*.

The i.MX6 performance counters do not allow exclusive access to just user or just kernel measurements. Also, i.MX6 SoC designers have unfortunately joined the PMU interrupts from all CPU cores, when ideally they should only be handled by the same CPU that raises them. You can start the i.MX6 with just one core, using the `maxcpus=1` kernel command-line argument, so that you can still use the `perf_events` interface.

To configure the Linux kernel to boot with one core, stop at the U-Boot prompt and change the `mmcargs` environment variable as follows:

```
> setenv mmcargs 'setenv bootargs console=${console},${baudrate}  
root=${mmcroot} ${extra_bootargs}; run videoargs'  
> setenv extra_bootargs maxcpus=1
```

 The mmcargs environmental variable is only used when booting from an MMC device like the microSD card. If the target is booting from another source, such as a network, the corresponding environmental variable will have to be changed. You can dump the whole U-Boot environment with the printenv U-Boot command, and change the required variable with setenv.

How to do it...

The interface introduces a `sys_perf_event_open()` syscall, with the counters being started and stopped using `ioctls`, and read either with `read()` calls or mmapping samples into circular buffers. The `perf_event_open()` syscall is defined as follows:

```
#include <linux/perf_event.h>
#include <linux/hw_breakpoint.h>

int perf_event_open(struct perf_event_attr *attr,
                    pid_t pid, int cpu, int group_fd,
                    unsigned long flags);
```

There is no C library wrapper for it, so it needs to be called using `syscall()`.

How it works...

Following is an example, `perf_example.c`, program modified from the `perf_event_open` man page to measure instruction count for a `printf` call:

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/ioctl.h>
#include <linux/perf_event.h>
#include <asm/unistd.h>

static long
perf_event_open(struct perf_event_attr *hw_event, pid_t pid,
               int cpu, int group_fd, unsigned long flags)
{
    int ret;

    ret = syscall(__NR_perf_event_open, hw_event, pid, cpu,
                 group_fd, flags);
```

```
        return ret;
    }

int
main(int argc, char **argv)
{
    struct perf_event_attr pe;
    long long count;
    int fd;

    memset(&pe, 0, sizeof(struct perf_event_attr));
    pe.type = PERF_TYPE_HARDWARE;
    pe.size = sizeof(struct perf_event_attr);
    pe.config = PERF_COUNT_HW_INSTRUCTIONS;
    pe.disabled = 1;

    fd = perf_event_open(&pe, 0, -1, -1, 0);
    if (fd == -1) {
        fprintf(stderr, "Error opening leader %llx\n", pe.config);
        exit(EXIT_FAILURE);
    }

    ioctl(fd, PERF_EVENT_IOC_RESET, 0);
    ioctl(fd, PERF_EVENT_IOC_ENABLE, 0);

    printf("Measuring instruction count for this printf\n");

    ioctl(fd, PERF_EVENT_IOC_DISABLE, 0);
    read(fd, &count, sizeof(long long));

    printf("Used %lld instructions\n", count);

    close(fd);

    return 0;
}
```

For compiling this program externally, we can use the following commands:

```
$ source /opt/poky/1.7.1/environment-setup-cortexa9hf-vfp-neon-poky-
linux-gnueabi
$ ${CC} perf_example.c -o perf_example
```

After copying the binary to your target, you can then execute it with the help of the following code:

```
# ./perf_example  
Measuring instruction count for this printf  
Used 0 instructions
```

Obviously, using zero instructions for the `printf()` call can't be correct. Looking into possible causes, we find a documented erratum (ERR006259) on i.MX6 processors that states that in order for the PMU to be used, the SoC needs to receive at least 4 JTAG clock cycles after power on reset.

Rerun the example with the JTAG connected:

```
# ./perf_example  
Measuring instruction count for this printf  
Used 3977 instructions
```

There's more...

Even though you can access the `perf_events` interface directly as in the preceding example, the recommended way to use it is through a user space application, such as `perf`, which we will see in the *Tracing and profiling with perf* recipe in this chapter.

See also

- ▶ The Technical Reference Manual at <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0388f/BEHGGDJC.html> for more information about the Cortex-A9 PMU

Using static kernel tracing

The Linux kernel is continuously being instrumented with static probe points called **tracepoints**, which when disabled have a very small overhead. They allow us to record more information than the function tracer we saw in *Chapter 2, The BSP Layer*. Tracepoints are used by multiple tracing and profiling tools in Yocto.

This recipe will explain how to use and define static tracepoints independently of user space tools.

Getting ready

Static tracepoints can be instrumented using custom kernel modules, and also through the event tracing infrastructure. Enabling any of the tracing features in the kernel will create a `/sys/kernel/debug/tracing/` directory; for example, the function tracing feature as explained in the *Using the kernel function tracing system* in Chapter 2, *The BSP Layer*.

So before continuing with this recipe, you need to configure the function tracing feature in the Linux kernel as explained before.

How to do it...

The static tracing functionality is exposed via the `debugfs` filesystem. The functionality offered by the interface includes:

- ▶ **Listing events:**

You can see a list of available tracepoints exposed via `sysfs` and ordered in subsystem directories with:

```
# ls /sys/kernel/debug/tracing/events/
asoc          ftrace        migrate      rCU          spi
block         gpio          module      regmap      sunrpc
cfg80211     header_event  napi       regulator   task
compaction    header_page   net        rpm         timer
drm           irq           oom        sched      udp
enable        jbd           power      scsi       vmscan
ext3          jbd2          printk     signal     workqueue
ext4          kmem          random    skb        writeback
filemap       mac80211    raw_syscalls  sock
```

Or in the `available_events` file with the `<subsystem>:<event>` format using the following commands:

```
# grep 'net'  /sys/kernel/debug/tracing/available_events
net:netif_rx
net:netif_receive_skb
net:net_dev_queue
net:net_dev_xmit
```

► **Describing events:**

Each event has a specific printing format that describes the information included in the log event, as follows:

```
#cat /sys/kernel/debug/tracing/events/net/netif_receive_skb/format
name: netif_receive_skb
ID: 378
format:
    field:unsigned short common_type;  offset:0;  size:2;
    signed:0;
    field:unsigned char common_flags;  offset:2;  size:1;
    signed:0;
    field:unsigned char common_preempt_count;  offset:3;
    size:1;  signed:0;
    field:int common_pid;  offset:4;  size:4;  signed:1;

    field:void * skbaddr;  offset:8;  size:4;  signed:0;
    field:unsigned int len;  offset:12;  size:4;  signed:0;
    field:_data_loc char[] name;  offset:16;  size:4;  signed:0;

print fmt: "dev=%s skbaddr=%p len=%u", __get_str(name), REC->skbaddr, REC->len
```

► **Enabling and disabling events:**

You can enable or disable events in the following ways:

- By echoing 0 or 1 to the event enable file:

```
# echo 1 >
    /sys/kernel/debug/tracing/events/net/netif_receive_skb/
enable
```

- By subsystem directory, which will enable or disable all the tracepoints in the directory/subsystem:

```
# echo 1 > /sys/kernel/debug/tracing/events/net/enable
```

- By echoing the unique tracepoint name into the set_event file:

```
# echo netif_receive_skb >>
    /sys/kernel/debug/tracing/set_event
```

Note the append operation >> is used not to clear events.

- ❑ Events can be disabled by appending an exclamation mark to their names:

```
# echo '!netif_receive_skb' >>
    /sys/kernel/debug/tracing/set_event
```

- ❑ Events can also be enabled/disabled by subsystem:

```
# echo 'net:*' > /sys/kernel/debug/tracing/set_event
```

- ❑ To disable all events:

```
# echo > /sys/kernel/debug/tracing/set_event
```

You can also enable tracepoints from boot by passing a `trace_event=<comma separated event list>` kernel command line-argument.

► **Adding events to the tracing buffer:**

To see the tracepoints appear on the tracing buffer, turn tracing on:

```
# echo 1 > /sys/kernel/debug/tracing/tracing_on
```

Tracepoint events are integrated into the `ftrace` subsystem so that if you enable a tracepoint, when a tracer is running, it will show up in the trace. Take a look at the following commands:

```
# cd /sys/kernel/debug/tracing
# echo 1 > events/net/netif_receive_skb/enable
# echo netif_receive_skb > set_ftrace_filter
# echo function > current_tracer
# cat trace
        <idle>-0      [000] ..s2  1858.542206:
netif_receive_skb <-napi_gro_receive
        <idle>-0      [000] ..s2  1858.542214:
netif_receive_skb: dev=eth0 skbaddr=dcb5bd80 len=168
```

How it works...

A tracepoint is inserted using the `TRACE_EVENT` macro. It inserts a callback in the kernel source that gets called with the tracepoint parameters as arguments. Tracepoints added with the `TRACE_EVENT` macro allow `ftrace` or any other tracer to use them. The callback inserts the trace at the calling tracer's ring buffer.

To insert a new tracepoint into the Linux kernel, define a new header file with a special format. By default, tracepoint kernel files are located in `include/trace/events`, but the kernel has functionality so that the header files can be located in a different path. This is useful when defining a tracepoint in a kernel module.

To use the tracepoint, the header file must be included in any file that inserts the tracepoint, and a single C file must define `CREATE_TRACE_POINTS`. For example, to extend the `Hello World` Linux kernel module we saw in a previous chapter with a tracepoint, add the following code to `meta-bsp-custom/recipes-kernel/hello-world-tracepoint/files/hello_world.c`:

```
#include <linux/module.h>
#include "linux/timer.h"
#define CREATE_TRACE_POINTS
#include "trace.h"

static struct timer_list hello_timer;

void hello_timer_callback(unsigned long data)
{
    char a[] = "Hello";
    char b[] = "World";
    printk("%s %s\n", a, b);
    /* Insert the static tracepoint */
    trace_log_dbg(a, b);
    /* Trigger the timer again in 8 seconds */
    mod_timer(&hello_timer, jiffies + msecs_to_jiffies(8000));
}

static int hello_world_init(void)
{
    /* Setup a timer to fire in 2 seconds */
    setup_timer(&hello_timer, hello_timer_callback, 0);
    mod_timer(&hello_timer, jiffies + msecs_to_jiffies(2000));
    return 0;
}

static void hello_world_exit(void)
{
    /* Delete the timer */
    del_timer(&hello_timer);
}

module_init(hello_world_init);
module_exit(hello_world_exit);

MODULE_LICENSE("GPL v2");
```

The tracepoint header file in `meta-bsp-custom/recipes-kernel/hello-world-tracepoint/files/trace.h` would be:

```
#undef TRACE_SYSTEM
#define TRACE_SYSTEM log_dbg

#if !defined(_HELLOWORLD_TRACE) || defined(TRACE_HEADER_MULTI_READ)
#define _HELLOWORLD_TRACE

#include <linux/tracepoint.h>

TRACE_EVENT(log_dbg,
            TP_PROTO(char *a, char *b),
            TP_ARGS(a, b),
            TP_STRUCT__entry(
                __string(a, a)
                __string(b, b)),
            TP_fast_assign(
                __assign_str(a, a);
                __assign_str(b, b);),
            TP_printk("log_dbg: a %s b %s",
                      __get_str(a), __get_str(b))
            );
#endif

/* This part must be outside protection */
#undef TRACE_INCLUDE_PATH
#undef TRACE_INCLUDE_FILE
#define TRACE_INCLUDE_PATH .
#define TRACE_INCLUDE_FILE trace
#include <trace/define_trace.h>
```

And the module's Makefile file in `meta-bsp-custom/recipes-kernel/hello-world-tracepoint/files/Makefile` would look as follows:

```
obj-m    := hello_world.o
CFLAGS_hello_world.o += -I$(src)

SRC := $(shell pwd)

all:
    $(MAKE) -C "$(KERNEL_SRC)" M=$(SRC)"

modules_install:
```

```
$ (MAKE) -C "$ (KERNEL_SRC)" M="$ (SRC)" modules_install

clean:
    rm -f *.o *~ core .depend *.cmd *.ko *.mod.c
    rm -f Module.markers Module.symvers modules.order
    rm -rf .tmp_versions Modules.symvers
```

Note the highlighted line that includes the current folder in the search path for include files.

We can now build the module externally, as we saw in the *Building external kernel modules* recipe in *Chapter 2, The BSP Layer*. The corresponding Yocto recipe is included in the source that accompanies the book. Here is the code for the same:

```
$ cd /opt/yocto/fsl-community-bsp/sources/meta-bsp-custom/recipes-
  kernel/hello-world-tracepoint/files/
$ source /opt/poky/1.7.1/environment-setup-cortexa9hf-vfp-neon-poky-
  linux-gnueabi
$ KERNEL_SRC=/opt/yocto/linux-wandboard make
```

After copying the resulting `hello_world.ko` module to the Wandboard's root filesystem, you can load it with:

```
# insmod hello_world.ko
Hello World
```

You can now see a new `log_dbg` directory inside `/sys/kernel/debug/tracing/events`, which contains a `log_dbg` event tracepoint with the following format:

```
# cat /sys/kernel/debug/tracing/events/log_dbg/log_dbg/format
name: log_dbg
ID: 622
format:
    field:unsigned short common_type;          offset:0;
size:2; signed:0;
    field:unsigned char common_flags;          offset:2;
size:1; signed:0;
    field:unsigned char common_preempt_count;   offset:3;
size:1; signed:0;
    field:int common_pid;      offset:4;        size:4; signed:1;

    field:__data_loc char[] a;      offset:8;        size:4;
signed:0;
```

```
    field:__data_loc char[] b;          offset:12;      size:4;
signed:0;

print fmt: "log_dbg: a %s b %s", __get_str(a), __get_str(b)
```

You can then enable the function tracer on the `hello_timer_callback` function:

```
# cd /sys/kernel/debug/tracing
# echo 1 > events/log_dbg/log_dbg/enable
# echo 1 > /sys/kernel/debug/tracing/tracing_on
# cat trace
<idle>-0      [000] ..s2      57.425040: log_dbg: log_dbg: a
Hello b World
```

There's more...

Static tracepoints can also be filtered. When an event matches a filter set, it is kept, otherwise it is discarded. Events without filters are always kept.

For example, to set a matching filter for the `log_dbg` event inserted in the preceding code, you could match either the `a` or `b` variables:

```
# echo "a == \"Hello\" >
/sys/kernel/debug/tracing/events/log_dbg/log_dbg/filter
```

See also

- ▶ The Linux kernel documentation at <https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/plain/Documentation/trace/events.txt> for more information regarding static tracepoints events
- ▶ The *Using the TRACE_EVENT() macro* article series by Steven Rostedt at <http://lwn.net/Articles/379903/>

Using dynamic kernel tracing

`kprobes` is a kernel debugging facility that allows us to dynamically break into almost any kernel function (except `kprobe` itself) to collect debugging and profiling information non-disruptively. Some architectures keep an array of blacklisted functions, which cannot be probed using `kprobe`, but on ARM the list is empty.

Because `kprobes` can be used to change a function's data and registers, it should only be used in development environments.

There are three types of probes:

- ▶ **kprobes**: This is the kernel probe which can be inserted into any location with more than one kprobe added at a single location, if needed.
- ▶ **jprobe**: This is the jumper probe inserted at the entry point of a kernel function to provide access to its arguments. Only one jprobe may be added at a given location.
- ▶ **kretprobe**: This is the return probe which triggers on a function return. Also, only one kretprobe may be added to the same location.

They are packaged into a kernel module, with the `init` function registering the probes and the `exit` function unregistering them.

This recipe will explain how to use all types of dynamic probes.

Getting ready

To configure the Linux kernel with kprobes support, you need to:

- ▶ Define the `CONFIG_KPROBES` configuration variable
- ▶ Define `CONFIG_MODULES` and `CONFIG_MODULE_UNLOAD` so that modules can be used to register probes
- ▶ Define `CONFIG_KALLSYMS` and `CONFIG_KALLSYMS_ALL` (recommended) so that kernel symbols can be looked up
- ▶ Optionally, define the `CONFIG_DEBUG_INFO` configuration variable so that probes can be inserted in the middle of functions as offsets from the entry point. To find the insertion point, you can use `objdump`, as seen in the following excerpt for the `do_sys_open` function:

```
arm-poky-linux-gnueabi-objdump -d -l vmlinux | grep
    do_sys_open
0010bfa8 <do_sys_open>:
do_sys_open():
0010c034: 0a000036      beq      0010c114
    <do_sys_open+0x16c>
0010c044: 1a000031      bne      0010c110
    <do_sys_open+0x168>
```

The `kprobes` API is defined in the `kprobes.h` file and includes registration/unregistration and enabling/disabling functions for the three types of probes as follows:

```
#include <linux/kprobes.h>
int register_kprobe(struct kprobe *kp);
int register_jprobe(struct jprobe *jp);
```

```
int register_kretprobe(struct kretprobe *rp);  
  
void unregister_kprobe(struct kprobe *kp);  
void unregister_jprobe(struct jprobe *jp);  
void unregister_kretprobe(struct kretprobe *rp);
```

By default, a kprobe probe is enabled when registering, except when the KPROBE_FLAG_DISABLED flag is passed. The following function definitions enable or disable the probe:

```
int disable_kprobe(struct kprobe *kp);  
int disable_kretprobe(struct kretprobe *rp);  
int disable_jprobe(struct jprobe *jp);  
  
int enable_kprobe(struct kprobe *kp);  
int enable_kretprobe(struct kretprobe *rp);  
int enable_jprobe(struct jprobe *jp);
```

The registered kprobe probes can be listed through debugfs:

```
$ cat /sys/kernel/debug/kprobes/list
```

They can globally be enabled or disabled with:

```
$ echo 0/1 > /sys/kernel/debug/kprobes/enabled
```

How to do it...

On registration, the kprobe probe places a breakpoint (or jump, if optimized) instruction at the start of the probed instruction. When the breakpoint is hit, a trap occurs, the registers are saved, and control passes to kprobes, which calls the pre-handler. It then single steps the breakpoint and calls the post-handler. If a fault occurs, the fault handler is called. Handlers can be NULL if desired.

A kprobe probe can be inserted either in a function symbol or into an address, using the offset field, but not in both.



On occasions, kprobe will still be too intrusive to debug certain problems, as it slows the functions and may affect scheduling and be problematic when called from interrupt context.

For example, to place a kprobe probe in the open syscall, we would use the `meta-bsp-custom/recipes-kernel/open-kprobe/files/kprobe_open.c` custom module:

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/kprobes.h>

static struct kprobe kp = {
    .symbol_name  = "do_sys_open",
};

static int handler_pre(struct kprobe *p, struct pt_regs *regs)
{
    pr_info("pre_handler: p->addr = 0x%p, lr = 0x%lx,"
            " sp = 0x%lx\n",
            p->addr, regs->ARM_lr, regs->ARM_sp);

    /* A dump_stack() here will give a stack backtrace */
    return 0;
}

static void handler_post(struct kprobe *p, struct pt_regs *regs,
                        unsigned long flags)
{
    pr_info("post_handler: p->addr = 0x%p, status = 0x%lx\n",
            p->addr, regs->ARM_cpsr);
}

static int handler_fault(struct kprobe *p, struct pt_regs *regs,
                        int trapnr)
{
    pr_info("fault_handler: p->addr = 0x%p, trap #%dn",
            p->addr, trapnr);
    /* Return 0 because we don't handle the fault. */
    return 0;
}

static int kprobe_init(void)
{
    int ret;
    kp.pre_handler = handler_pre;
    kp.post_handler = handler_post;
```

```
kp.fault_handler = handler_fault;

ret = register_kprobe(&kp);
if (ret < 0) {
    pr_err("register_kprobe failed, returned %d\n", ret);
    return ret;
}
pr_info("Planted kprobe at %p\n", kp.addr);
return 0;
}

static void kprobe_exit(void)
{
    unregister_kprobe(&kp);
    pr_info("kprobe at %p unregistered\n", kp.addr);
}

module_init(kprobe_init)
module_exit(kprobe_exit)
MODULE_LICENSE("GPL");
```

We compile it with a Yocto recipe, as explained in the *Building external kernel modules* recipe in *Chapter 2, The BSP Layer*. Here is the code for the `meta-bsp-custom/recipes-kernel/open-kprobe/open-kprobe.bb` Yocto recipe file:

```
SUMMARY = "kprobe on do_sys_open kernel module."
LICENSE = "GPLv2"
LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/GPL-2.0;md5=801f80980d171dd6425610833a22dbe6"

inherit module

PV = "0.1"

SRC_URI = " \
    file://kprobe_open.c \
    file://Makefile \
"
S = "${WORKDIR}"
```

With the `Makefile` file in `meta-bsp-custom/recipes-kernel/open-kprobe/files/` `Makefile` being:

```
obj-m := kprobe_open.o

SRC := $(shell pwd)

all:
$(MAKE) -C "$(KERNEL_SRC)" M=$(SRC)

modules_install:
$(MAKE) -C "$(KERNEL_SRC)" M=$(SRC) modules_install

clean:
rm -f *.o *~ core .depend *.cmd *.ko *.mod.c
rm -f Module.markers Module.symvers modules.order
rm -rf .tmp_versions Modules.symvers
```

Copy it to a target running the same kernel it has been linked against, and load it with the following:

```
$ insmod kprobe_open.ko
Planted kprobe at 8010da84
```

We can now see the handlers printing in the console when a file is opened:

```
pre_handler: p->addr = 0x8010da84, lr = 0x8010dc34, sp = 0xdca75f98
post_handler: p->addr = 0x8010da84, status = 0x80070013
```

There's more...

A jprobe probe is implemented with a kprobe. It sets a breakpoint at the given symbol or address (but it must be the first instruction of a function), and makes a copy of a portion of the stack. When hit, it then jumps to the handler with the same registers and stack as the probed function. The handler must have the same argument list and return type as the probed function, and call `jprobe_return()` before returning to pass the control back to kprobes. Then the original stack and CPU state are restored and the probed function is called.

Following is an example of a jprobe in the open syscall in the `meta-bsp-custom/recipes-kernel/open-jprobe/files/jprobe_open.c` file:

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/kprobes.h>
```

```
static long jdo_sys_open(int dfd, const char __user *filename, int
    flags, umode_t mode)
{
    pr_info("jprobe: dfd = 0x%x, filename = 0xs "
        "flags = 0x%x mode umode %x\n", dfd, filename, flags, mode);

    /* Always end with a call to jprobe_return(). */
    jprobe_return();
    return 0;
}

static struct jprobe my_jprobe = {
    .entry          = jdo_sys_open,
    .kp = {
        .symbol_name  = "do_sys_open",
    },
};

static int jprobe_init(void)
{
    int ret;

    ret = register_jprobe(&my_jprobe);
    if (ret < 0) {
        pr_err("register_jprobe failed, returned %d\n", ret);
        return -1;
    }
    pr_info("Planted jprobe at %p, handler addr %p\n",
            my_jprobe.kp.addr, my_jprobe.entry);
    return 0;
}

static void jprobe_exit(void)
{
    unregister_jprobe(&my_jprobe);
    pr_info("jprobe at %p unregistered\n", my_jprobe.kp.addr);
}

module_init(jprobe_init)
module_exit(jprobe_exit)
MODULE_LICENSE("GPL");
```

A kretprobe probe sets a kprobe at the given symbol or function address which when hit, replaces the return address with a trampoline, usually a nop instruction, where kprobe is registered. When the probed function returns, the kprobe probe on the trampoline is hit, calling the return handler and setting back the original return address before resuming execution.

Following is an example of a kretprobe probe in the open syscall in the `meta-bsp-custom/recipes-kernel/open-kretprobe/files/kretprobe_open.c` file:

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/kprobes.h>
#include <linux/ktime.h>
#include <linux/limits.h>
#include <linux/sched.h>

/* per-instance private data */
struct my_data {
    ktime_t entry_stamp;
};

static int entry_handler(struct kretprobe_instance *ri, struct pt_regs *regs)
{
    struct my_data *data;

    if (!current->mm)
        return 1; /* Skip kernel threads */

    data = (struct my_data *)ri->data;
    data->entry_stamp = ktime_get();
    return 0;
}

static int ret_handler(struct kretprobe_instance *ri, struct pt_regs *regs)
{
    int retval = regs_return_value(regs);
    struct my_data *data = (struct my_data *)ri->data;
    s64 delta;
    ktime_t now;

    now = ktime_get();
    delta = ktime_to_ns(ktime_sub(now, data->entry_stamp));
    pr_info("returned %d and took %lld ns to execute\n",
            retval, delta);
```

```
        retval, (long long)delta);
    return 0;
}

static struct kretprobe my_kretprobe = {
    .handler      = ret_handler,
    .entry_handler      = entry_handler,
    .data_size      = sizeof(struct my_data),
    .maxactive      = 20,
};

static int kretprobe_init(void)
{
    int ret;

    my_kretprobe.kp.symbol_name = "do_sys_open";
    ret = register_kretprobe(&my_kretprobe);
    if (ret < 0) {
        pr_err("register_kretprobe failed, returned %d\n",
               ret);
        return -1;
    }
    pr_info("Planted return probe at %s: %p\n",
            my_kretprobe.kp.symbol_name,
            my_kretprobe.kp.addr);
    return 0;
}

static void kretprobe_exit(void)
{
    unregister_kretprobe(&my_kretprobe);
    pr_info("kretprobe at %p unregistered\n",
            my_kretprobe.kp.addr);

    /* nmissed > 0 suggests that maxactive was set too low. */
    pr_info("Missed probing %d instances of %s\n",
            my_kretprobe.nmissed, my_kretprobe.kp.symbol_name);
}

module_init(kretprobe_init)
module_exit(kretprobe_exit)
MODULE_LICENSE("GPL");
```

The highlighted `maxactive` variable is the number of reserved storage for return addresses in the `kretprobe` probe, and by default, it is the number of CPUs (or twice the number of CPUs in preemptive systems with a maximum of 10). If `maxactive` is too low, some probes will be missed.

The complete examples, including Yocto recipes, can be found in the source that accompanies the book.

See also

- ▶ The kprobes documentation on the Linux kernel at <https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/kprobes.txt>

Using dynamic kernel events

Although dynamic tracing is a very useful feature, custom kernel modules is not a user-friendly interface. Fortunately, the Linux kernel has been extended with the support of `kprobe` events, which allow us to set `kprobes` probes using a `debugfs` interface.

Getting ready

To make use of this feature, we need to configure our kernel with the `CONFIG_KPROBE_EVENT` configuration variable.

How to do it...

The `debugfs` interface adds probes via the `/sys/kernel/debug/tracing/kprobe_events` file. For example, to add a `kprobe` called `example_probe` to the `do_sys_open` function, you can execute the following command:

```
# echo 'p:example_probe do_sys_open dfd=%r0 filename=%r1 flags=%r2 mode=%r3' > /sys/kernel/debug/tracing/kprobe_events
```

The probe will print the function's argument list, according to the function's declaration arguments as seen in the function's definition below:

```
long do_sys_open(int dfd, const char __user *filename, int flags, umode_t mode);
```

You can then manage kprobes through the sysfs as follows:

- ▶ To see all the registered probes:

```
# cat /sys/kernel/debug/tracing/kprobe_events
p:kprobes/example_probe do_sys_open dfd=%r0 filename=%r1
flags=%r2 mode=%r3
```

- ▶ To print the probe format:

```
# cat
/sys/kernel/debug/tracing/events/kprobes/example_probe/format
name: example_probe
ID: 1235
format:
    field:unsigned short common_type;          offset:0;
size:2; signed:0;
    field:unsigned char common_flags;         offset:2;
size:1; signed:0;
    field:unsigned char common_preempt_count;
offset:3;      size:1; signed:0;
    field:int common_pid;      offset:4;        size:4;
signed:1;
    field:unsigned long __probe_ip; offset:8;
size:4; signed:0;
    field:u32 dfd;    offset:12;       size:4; signed:0;
    field:u32 filename;     offset:16;       size:4;
signed:0;
    field:u32 flags;        offset:20;       size:4;
signed:0;
    field:u32 mode; offset:24;       size:4; signed:0;
```

```
print fmt: "(%lx) dfd=%lx filename=%lx flags=%lx mode=%lx",
REC->__probe_ip, REC->dfd, REC->filename, REC->flags, REC-
>mode
```

- ▶ To enable the probe use the following command:

```
# echo 1 >
/sys/kernel/debug/tracing/events/kprobes/example_probe/enable
```

- ▶ To see the probe output on either the trace or trace_pipe files:

```
# cat /sys/kernel/debug/tracing/trace
# tracer: nop
#
```

```
# entries-in-buffer/entries-written: 59/59 #P:4
#
#           _-----> irqs-off
#           / _----> need-resched
#           | / _---> hardirq/softirq
#           || / _--> preempt-depth
#           ||| /      delay
#           TASK-PID    CPU#  ||||  TIMESTAMP  FUNCTION
#           | |        |  ||||  |          |
sh-737  [000] d...  1610.378856: example_probe:
(do_sys_open+0x0/0x184) dfd=fffffff9c filename=f88488
flags=20241 mode=16
sh-737  [000] d...  1660.888921: example_probe:
(do_sys_open+0x0/0x184) dfd=fffffff9c filename=f88a88
flags=20241 mode=16
```

- ▶ To clear the probe (after disabling it):

```
# echo '-:example_probe' >>
/sys/kernel/debug/tracing/kprobe_events
```

- ▶ To clear all probes:

```
# echo > /sys/kernel/debug/tracing/kprobe_events
```

- ▶ To check the number of hit and missed events:

```
# cat /sys/kernel/debug/tracing/kprobe_profile
example_probe          78          0
```

With the format being as follows:

```
<event name> <hits> <miss-hits>
```

How it works...

To set a probe we use the following syntax:

```
<type>:<event name> <symbol> <fetch arguments>
```

Let's explain each of the mentioned parameters:

- ▶ `type`: This is either `p` for kprobe or `r` for a return probe.
- ▶ `event name`: This is optional and has the format `<group/event>`. If the group name is omitted, it defaults to `kprobes`, and if the event name is omitted, it is autogenerated based on the symbol. When an event name is given, it adds a directory under `/sys/kernel/debug/tracing/events/kprobes/` with the following content:
 - `id`: This is the ID of the probe event
 - `filter`: This specifies user filtering rules
 - `format`: This is the format of the probe event
 - `enabled`: This is used to enable or disable the probe event
- ▶ `symbol`: This is either the symbol name plus an optional offset or the memory address where the probe is to be inserted.
- ▶ `fetch arguments`: These are optional and represent the information to extract with a maximum of 128 arguments. They have the following format:

`<name>=<offset>(<argument>):<type>`

Lets explain each of the mentioned parameters:

- `name`: This sets the argument name
- `offset`: This adds an offset to the address argument
- `argument`: This can be of the following format:

`%<register>`: This fetches the specified register. For ARM these are:

`r0 to r10`
`fp`
`ip`
`sp`
`lr`
`pc`
`cpsr`
`ORIG_r0`

`@<address>`: This fetches the memory at the specified kernel address

`@<symbol><offset>`: This fetches the memory at the specified symbol and optional offset

`$stack`: This fetches the stack address

`$stack<N>`: This fetches the *n*th entry of the stack

And for return probes we have:

`$retval`: This fetches the return value

- ❑ `type`: This one sets the argument type used by kprobe to access the memory from the following options:

`u8,u16,u32,u64`, for unsigned types

`s8,s16,s32,s64`, for signed types

`string`, for null terminated strings

`bitfield`, with the following format:

`b<bit-width>@<bit-offset>/<container-size>`

There's more...

Current versions of the Linux kernel (from v3.14 onwards) also have support for user space probe events (uprobes), with a similar interface to the one for the kprobes events.

Exploring Yocto's tracing and profiling tools

Tracing and profiling tools are used to increase the performance, efficiency, and quality of both, applications and systems. User space tracing and profiling tools make use of performance counters and static and dynamic tracing functionality that the Linux kernel offers, as we have seen in the previous recipes.

Getting ready

Tracing enables us to log an application's activity so that its behavior can be analyzed, optimized, and corrected.

Yocto offers several tracing tools including:

- ▶ **trace-cmd**: This is a command line interface to the ftrace kernel subsystem, and **kernelshark**, a graphical interface to trace-cmd.
- ▶ **perf**: This is a tool that originated in the Linux kernel as a command line interface to its performance counter events subsystem. It has since then expanded and added several other tracing mechanisms.

- ▶ **blktrace:** This is a tool that provides information about the block layer input/output.
- ▶ **Linux Trace Toolkit Next Generation (LTTng):** This is a tool that allows for correlated tracing of the Linux kernel, applications, and libraries. Yocto also includes **babeltrace**, a tool to translate the traces into human readable logs.
- ▶ **SystemTap:** This is a tool to dynamically instrument the Linux kernel.

Profiling refers to a group of techniques used to measure an application's consumed resources and the time taken to execute an application. The data is then used to improve the application's performance and optimize it. Some of the aforementioned tools such as perf and SystemTap have evolved to become powerful tracing and profiling tools.

Apart from the enlisted tracing tools, which can also be used for profiling, Yocto offers several other profiling tools:

- ▶ **OProfile:** This is a statistical profiler for Linux that profiles all running code with low overhead.
- ▶ **Powertop:** This is a tool used to analyze the system's power consumption and power management.
- ▶ **Latencytop:** This is a tool used to analyze system latencies.
- ▶ **Sysprof:** This tool is included for Intel architectures on X11 graphical images. It does not work on ARM architectures.

How to do it...

These tools can be added to your target image either individually or with the `tools-profile` feature. To use the tools, we also need to include debug information in our applications. To this extent we should use the `-dbg` version of the packages, or better, configure Yocto so that debug information is generated with the `dbg-pkgs` image feature. To add both features to your images, add the following to your project's `conf/local.conf` file:

```
EXTRA_IMAGE_FEATURES = "tools-profile dbg-pkgs"
```

The `-sdk` version of target images already adds these features.

There's more...

Apart from these tools, Yocto also offers the standard monitoring tools available on a Linux system. Some examples are:

- ▶ **htop:** This tool is available in the `meta-oe` layer and provides process monitoring.
- ▶ **iostat:** This tool is also included in the `meta-oe` layer and provides block device I/O statistics by process.

- ▶ **procps:** This one is available in Poky and includes the following tools:
 - **ps:** This tool is used to list and provide process statuses.
 - **vmstat:** This is used for virtual memory statistics.
 - **uptime:** This is useful for load averages monitoring.
 - **free:** This is used for memory usage monitoring. Remember to take kernel caches into account.
 - **slabtop:** This one provides memory usage statistics for the kernel slab allocator.
- ▶ **sysstat:** This is available in Poky and contains, among others, the following tools:
 - **pidstat:** This is another option for process statistics.
 - **iostat:** This one provides block I/O statistics.
 - **mpstat:** This tool provides multi-processor statistics.

And Yocto also offers the following network tools:

- ▶ **tcpdump:** This networking tool is included in the `meta-networking` layer in `meta-openembedded`. It captures and analyzes network traffic.
- ▶ **netstat:** This is part of the `net-tools` package in Poky. It provides network protocol statistics.
- ▶ **ss:** This tool is included in the `iproute2` package in Poky. It provides sockets statistics.

Tracing and profiling with perf

The `perf` Linux tool can instrument the Linux kernel with both hardware and software performance counter events as well as static and dynamic kernel trace points. For this, it uses the kernel functionality we have seen in previous recipes, providing a common interface to all of them.

This tool can be used to debug, troubleshoot, optimize, and measure applications, workloads, or the full system, which covers the processor, kernel, and applications. `Perf` is probably the most complete of the tracing and profiling tools available for a Linux system.

Getting ready

The `perf` source is part of the Linux kernel. To include `perf` in your system, add the following to your `conf/local.conf` file:

```
IMAGE_INSTALL_append = " perf"
```

Perf is also part of the `tools-profile` image feature, so you can also add it with the following:

```
EXTRA_IMAGE_FEATURES += "tools-profile"
```

Perf is also included in the `-sdk` images.

To take the maximum advantage of this tool, we need to have symbols both in user space applications and libraries, as well as the Linux kernel. For this, we need to avoid stripping binaries by adding the following to the `conf/local.conf` configuration file:

```
INHIBIT_PACKAGE_STRIP = "1"
```

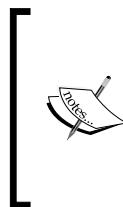
Also, adding the debug information of the applications by adding the following is recommended:

```
EXTRA_IMAGE_FEATURES += "dbg-pkgs"
```

By default, the debug information is placed in a `.debug` directory in the same location as the binary it corresponds to. But perf needs a central location to look for all debug information. So, to configure our debug information with a structure that perf understands, we also need the following in our `conf/local.conf` configuration file:

```
PACKAGE_DEBUG_SPLIT_STYLE = 'debug-file-directory'
```

Finally, configure the Linux kernel with the `CONFIG_DEBUG_INFO` configuration variable to include debug information, `CONFIG_KALLSYMS` to add debug symbols into the kernel, and `CONFIG_FRAME_POINTER` to be able to see complete stack traces.



As we saw in the *Using the kernel's performance counters* recipe, we will also need to pass `maxcpus=1` (or `maxcpus=0` to disable SMP) to the Linux kernel in order to use the i.MX6 PMU, due to the sharing of the PMU interrupt between all cores. Also, in order to use the PMU on i.MX6 processors, the SoC needs to receive at least 4 JTAG clock cycles after power on reset. This is documented in the errata number *ERR006259*.

At the time of writing, the `meta-fsl-arm` layer for Yocto 1.7 disables some of perf features. To be able to follow the upcoming examples, remove the following line from the `meta-fsl-arm` layer's `/opt/yocto/fsl-community-bsp/sources/meta-fsl-arm/conf/machine/include/imx-base.inc` file:

```
-PERF_FEATURES_ENABLE = ""
```

Newer Yocto releases will include this by default.

How to do it...

Perf can be used to provide a default set of event statistics for a particular workload with:

```
# perf stat <command>
```

For example, a single ping will provide the following output:

```
# perf stat ping -c 1 192.168.1.1
PING 192.168.1.1 (192.168.1.1): 56 data bytes
64 bytes from 192.168.1.1: seq=0 ttl=64 time=6.489 ms

--- 192.168.1.1 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 6.489/6.489/6.489 ms
```

Performance counter stats for 'ping -c 1 192.168.1.1':

```
 8.984333 task-clock          # 0.360 CPUs utilized
      15 context-switches       # 0.002 M/sec
        0 cpu-migrations        # 0.000 K/sec
      140 page-faults          # 0.016 M/sec
  3433188 cycles              # 0.382 GHz
  123948 stalled-cycles-frontend # 3.61% frontend
cycles idle
  418329 stalled-cycles-backend # 12.18% backend
cycles idle
  234497 instructions         # 0.07 insns per
cycle
                                # 1.78 stalled
cycles per insn
  22649 branches              # 2.521 M/sec
    8123 branch-misses        # 35.86% of all
branches

 0.024962333 seconds time elapsed
```

If we are only interested in a particular set of events, we can specify the events we want to output information from using the `-e` option.

We can also sample data and store it so that it can be later analyzed:

```
# perf record <command>
```

Better still, we can add stack backtraces with the -g option:

```
# perf record -g -- ping -c 1 192.168.1.1
```

The result will be stored on a `perf.data` file which we would then analyze with:

```
# perf report
```

Its output can be seen in the following screenshot:

```
Samples: 22 of event 'cycles', Event count (approx.): 2307629
+ 20.76% ping [kernel.kallsyms] [k] queue_work_on
+ 18.23% ping [kernel.kallsyms] [k] do_page_fault
+ 10.02% ping [kernel.kallsyms] [k] __percpu_counter_add
+ 9.80% ping [kernel.kallsyms] [k] handle_mm_fault
+ 8.26% ping ld-2.19.so [.] __udivsi3
+ 7.59% ping [kernel.kallsyms] [k] filemap_fault
+ 6.82% ping [kernel.kallsyms] [k] __memzero
+ 5.95% ping ld-2.19.so [.] open_verify
+ 4.96% ping [kernel.kallsyms] [k] __sync_icache_dcache
+ 3.84% ping ld-2.19.so [.] _dl_start
+ 2.58% ping [kernel.kallsyms] [k] padzero
+ 1.02% ping [kernel.kallsyms] [k] do_brk
+ 0.15% ping [kernel.kallsyms] [k] mprotect_fixup
+ 0.03% ping [kernel.kallsyms] [k] perf_event_comm
```

The functions order may be customized with the `--sort` option.

We can see how `perf` has resolved both user space and kernel symbols. `Perf` will read kernel symbols from the Linux kernel ELF file under `/boot`. If it is stored in a non-standard location, we can optionally pass its location with a `-k` option. If it does not find it, it will fall back to using `/proc/kallsyms`, where the Linux kernel exports the kernel symbols to user space when built with the `CONFIG_KALLSYMS` configuration variable.

If a `perf` report is not showing kernel symbols, it may be because the ELF file does not match the running kernel. You can try to rename it and see if using `/proc/kallsyms` works.

Also, to obtain complete backtraces, applications need to be compiled with debug optimization by using the `DEBUG_BUILD` configuration variable, as we saw earlier in this chapter.

By default, Perf uses a **newt** interface (TUI) that needs the `expand` utility, part of coreutils. If coreutils is not included in your root filesystem, you can ask for a text-only output with:

```
# perf report -stdio
```

After executing the preceding command we get the following output:

```
=====
# captured on: Fri Mar  6 21:34:41 2015
# hostname : ccimx6sbc
# os release : 3.10.54-dey+g8f13306f52e0
# perf version : 3.10.54
# arch : armv7l
# cores online : 4
# ncores avail : 1
# cpudesc : (null)
# total memory : 0 kB
# cmdline :
# event : name = cycles, type = 0, config = 0x0, config1 = 0x0, config2 = 0x0, excl_usr = 0, excl_kern = 0, excl_host = 0, excl_guest = 1, precise_ip = 0
# pau mappings: not available
=====

# Samples: 22 of event 'cycles'
# Event count (approx.): 2307629

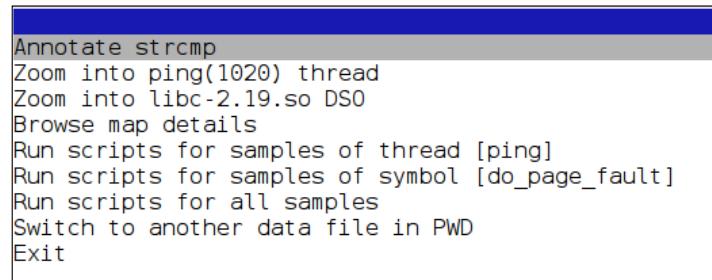
Overhead Command Shared Object Symbol
-----
[31m 20.76% [m ping [kernel.kallsyms] [k] queue_work_on
| ... queue_work_on
|   ... queue_work_on
|     ... tty_flip_buffer_push
|       ... pty_write
|         ... do_tty_output
|           ... process_output
|             ... n_tty_write
|               ... tty_write
|                 ... vts_write
|                   ... Sys_write
|                     ... ret_fast_syscall
|                       ... _GL__libc_write
|                         ...
|                           [31m--50.86%- [m0x676e6970
|                           [31m-49.14%- [m[...]
[31m 18.23% [m ping [kernel.kallsyms] [k] do_page_fault
| ... do_page_fault
|   ... do_DataAbort
|     ... doabt_usr
|       ...
|           [31m--51.41%- [mstrcmp
|           [31m-48.59%- [m_dl_addr
```

We can see all the functions called with the following columns:

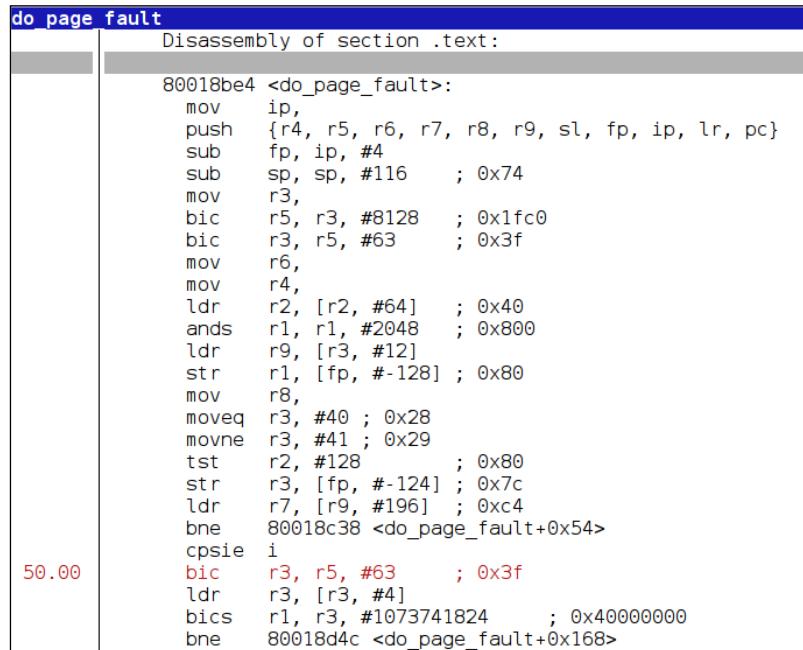
- ▶ **Overhead:** This represents the percentage of the sampling data corresponding to that function.
- ▶ **Command:** This refers to the name of the command passed to the perf record.
- ▶ **Shared Object:** This represents the ELF image name (`kernel.kallsyms` will appear for the kernel).
- ▶ **Privilege Level:** It has the following modes:
 - for user mode
 - `k` for kernel mode
 - `g` for virtualized guest kernel
 - `u` for virtualized host user space
 - `H` for hypervisor
- ▶ **Symbol:** This is the resolved symbol name.

Debugging, Tracing, and Profiling

In the TUI interface, we can press enter on a function name to access a sub-menu, which will give us the following output:



From this we can, for example, annotate the code as shown in the following screenshot:



If using text mode, we can also get annotated output with:

```
# perf annotate -d <command>
```

Perf can also do system-wide profiling instead of focusing on a specific workload. For example, to monitor the system for five seconds, we would execute the following command:

```
# perf stat -a sleep 5  
Performance counter stats for 'sleep 5':
```

```

      5006.660002 task-clock          # 1.000 CPUs
utilized[100.00%]

      324 context-switches         # 0.065 K/sec
[100.00%]

      0 cpu-migrations           # 0.000 K/sec
[100.00%]

      126 page-faults            # 0.025 K/sec

      12200175 cycles             # 0.002 GHz [100.00%]
cycles idle      [100.00%]

      2844703 stalled-cycles-frontend # 23.32% frontend
cycles idle      [100.00%]

      9152564 stalled-cycles-backend # 75.02% backend
cycles idle      [100.00%]

      4645466 instructions        # 0.38 insns per
cycle

      # 1.97 stalled
cycles per insn [100.00%]

      479051 branches             # 0.096 M/sec
[100.00%]

      222903 branch-misses        # 46.53% of all
branches

5.006115001 seconds time elapsed

```

Or to sample the system for five seconds, we will execute the following command:

```
# perf record -a -g -- sleep 5
```

When using system-wide measurements the command is just used as measurement duration. For this, the `sleep` command will not consume extra cycles.

How it works...

The `perf` tool provides statistics for both user and kernel events occurring in the system. It can instrument in two modes:

- ▶ **Event counting** (`perf stat`): This counts events in kernel context and prints statistics at the end. It has the least overhead.
- ▶ **Event sampling** (`perf record`): This writes the gathered data to a file at a given sampling period. The data can then be read as profiling (`perf report`) or trace data (`perf script`). Gathering data to a file can be resource intensive and the file can quickly grow in size.

By default, perf counts events for all the threads in the given command, including child processes, until the command finishes or is interrupted.

A generic way to run perf is as follows:

```
perf stat|record [-e <comma separated event list> --filter '<expr>']  
[-o <filename>] [--] <command> [<arguments>]
```

Let's explain the preceding code in detail:

- ▶ e: This specifies an event list to use instead of the default set of events. An event filter can also be specified, with its syntax explained in the Linux kernel source documentation at Documentation/trace/events.txt.
- ▶ o: This specifies the output file name, by default perf.data.
- ▶ --: This is used as a separator when the command needs arguments.

It can also start or sample a running process by passing the -p <pid> option.

We can obtain a list of all available events by executing the following command:

```
# perf list
```

Or on a specific subsystem with the following command:

```
# perf list '<subsystem>:'*
```

You can also access raw PMU events directly by using the r<event> event, for example, to read the data cache misses on an ARM core:

```
# perf stat -e r3 sleep 5
```

Unless specified, the perf record will sample hardware events at an average rate of 1000 Hz, but the rate can be modified with the -F <freq> argument. Tracepoints will be counted on each occurrence.

Reading tracing data

Perf records samples and stores tracing data in a file. The raw timestamped trace data can be seen with:

```
# perf script
```

After executing the command we get the following output:

```
# =====
# captured on: Fri Mar  6 21:44:19 2015
# hostname : ccimx6sbc
# os release : 3.10.54-dey+g8f13306f52e0
# perf version : 3.10.54
# arch : armv7l
# nr_cpus online : 4
# nr_cpus avail : 1
# cpuidle : (null)
# total memory : 0 kB
# cadline :
# event : name = cycles, type = 0, config = 0x0, config1 = 0x0, config2 = 0x0, excl_usr = 0, excl_kern = 0, excl_host = 0, excl_guest = 1, precise_ip = 0
# pmu mappings: not available
# =====
perf 1045 [000] 620.536754: cycles:
 8000c6900 mp_call_function_single ([kernel.kallsyms])
 800083e80 cpu_function_call ([kernel.kallsyms])
 800083e80 perf_event_enable ([kernel.kallsyms])
 800084260 perf_event_for_each_child ([kernel.kallsyms])
 800086104 perf_ioctl ([kernel.kallsyms])
 800d9ac4 do_vfs_ioctl ([kernel.kallsyms])
 800d9cdc sys_ioctl ([kernel.kallsyms])
 8000e480 ret_fast_syscall ([kernel.kallsyms])
 769bc8cc _GI__ioctl (/lib/libc-2.19.so)
 7eafcadc [unknown] ([unknown])

perf 1045 [000] 620.536770: cycles:
 8000c6900 mp_call_function_single ([kernel.kallsyms])
 800083e80 cpu_function_call ([kernel.kallsyms])
 800083e80 perf_event_enable ([kernel.kallsyms])
 800084260 perf_event_for_each_child ([kernel.kallsyms])
 800086104 perf_ioctl ([kernel.kallsyms])
 800d9ac4 do_vfs_ioctl ([kernel.kallsyms])
 800d9cdc sys_ioctl ([kernel.kallsyms])
 8000e480 ret_fast_syscall ([kernel.kallsyms])
 769bc8cc _GI__ioctl (/lib/libc-2.19.so)
 7eafcadc [unknown] ([unknown])
```

As we have seen, we can use a perf report to look at the sampled data formatted for profiling analysis, but we can also generate python scripts that we can then modify to change the way the data is presented, by running the following line of code:

```
# perf script -g python
```

This will generate a `perf-script.py` script that looks as follows:

```
import os
import sys

sys.path.append(os.environ['PERF_EXEC_PATH'] + \
    '/scripts/python/Perf-Trace-Util/lib/Perf/Trace')

from perf_trace_context import *
from Core import *

def trace_begin():
    print "in trace_begin"

def trace_end():
    print "in trace_end"

def trace_unhandled(event_name, context, event_fields_dict):
    print ' '.join(['%s=%s'%(k,str(v))for k,v in sorted(event_fields_dict.items())])

def print_header(event_name, cpu, secs, nsecs, pid, comm):
    print "%-20s %5u %05u.%09u %8u %-20s" % \
        (event_name, cpu, secs, nsecs, pid, comm),
```

To run the script, use the following command:

```
# perf script -s perf-script.py
```

You need to install the `perf-python` package in our target image. You can add this to your image with:

```
IMAGE_INSTALL_append = " perf-python"
```

Now you will get a similar output as with the `perf` script earlier. But now you can modify the print statements in the python code to post process the sampled data to your specific needs.

There's more...

Perf can use dynamic events to extend the event list to any location where `kprobe` can be placed. For this, configure the kernel for `kprobe` and `uprobe` support (if available), as seen in the *Using dynamic kernel events* recipe earlier.

To add a probe point in a specific function execute the following command:

```
# perf probe --add "tcp_sendmsg"
Added new event:
probe:tcp_sendmsg      (on tcp_sendmsg)
```

You can now use it in all perf tools, such as profiling the download of a file:

```
# perf record -e probe:tcp_sendmsg -a -g -- wget
http://downloads.yoctoproject.org/releases/yocto/yocto-
1.7.1/RELEASENOTES
Connecting to downloads.yoctoproject.org (198.145.29.10:80)
RELEASENOTES          100% | ****
***** | 11924    0:00:00 ETA
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.025 MB perf.data (~1074 samples)
 ]
```

And you can view the profiling data executing the following command:

```
# perf report
```

And then you get the following output:

```
Samples: 14 of event 'probe:tcp_sendmsg', Event count (approx.): 14
- 92.86% dropbear [kernel.kallsyms] [k] tcp_sendmsg
  tcp_sendmsg
  sock_aio_write
  do_sync_ready_writev
  do_ready_writev
  vfs_writev
  Sys_writev
  ret_fast_syscall
  __libc_writev
  0
- 7.14%     wget [kernel.kallsyms] [k] tcp_sendmsg
  tcp_sendmsg
  sock_aio_write
  do_sync_write
  vfs_write
  Sys_write
  ret_fast_syscall
  __GI__libc_write
```



You may need to configure DNS servers in your target for the wget command as seen in the preceding code to work. To use Google's public DNS servers, you can add the following to your /etc/resolv.conf file:

```
nameserver 8.8.8.8
nameserver 8.8.4.4
```

You can then delete the probe with:

```
# perf probe --del tcp_sendmsg
/sys/kernel/debug//tracing/uprobe_events file does not exist - please
rebuild kernel with CONFIG_UPROBE_EVENT.

Removed event: probe:tcp_sendmsg
```

Profile charts

System behavior can be visualized using a perf timechart. To gather data, run:

```
# perf timechart record -- <command> <arguments>
```

And to turn it into an svg file use the following command:

```
# perf timechart
```

Using perf as strace substitute

Perf can be used as an alternative to strace but with much less overhead with the following syntax:

```
# perf trace record <command>
```

However, the Yocto recipe for perf does not currently build this support. We can see the missing library in the compilation log:

```
Makefile:681: No libaudit.h found, disables 'trace' tool, please  
install audit-libs-devel or libaudit-dev
```

See also

- ▶ A list of the available ARM i.MX6 PMU events at <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0388f/BEHGGDJC.html>
- ▶ An extended tutorial in the use of perf at <https://perf.wiki.kernel.org/index.php/Tutorial>
- ▶ Some advanced examples at Brendan Gregg's perf site <http://www.brendangregg.com/perf.html>

Using SystemTap

SystemTap is a GPLv2 licensed system wide tool that allows you to gather tracing and profiling data from a running Linux system. The user writes a `systemtap` script, which is then compiled into a Linux kernel module linked against the same kernel source it is going to run under.

The script sets events and handlers, which are called by the kernel module on the specified events triggering. For this, it uses the `kprobes` and `uprobes` (if available) interfaces in the kernel, as we saw in the *Using dynamic kernel events* recipe before.

Getting ready

To use SystemTap, we need to add it to our target image either by adding it specifically, as in:

```
IMAGE_INSTALL_append = " systemtap"
```

We can also add it by using the `tools-profile` image feature, or an `-sdk` image.

We will also need an SSH server running on the target. This is already available on the `-sdk` image; otherwise we can add one to our image with the following:

```
EXTRA_IMAGE_FEATURES += "ssh-server-openssh"
```

We will also need to compile the kernel with the `CONFIG_DEBUG_INFO` configuration variable to include debug information, as well as performance events counters and kprobes as explained in previous recipes.

How to do it...

To use systemtap on a Yocto system, we need to run the `crosstab` utility in the host, passing it the `systemtap` script to run. For example, to run the `sys_open.stp` sample script, we can run the following code:

```
probe begin
{
    print("Monitoring starts\n")
    printf("%6s %6s %16s\n", "UID", "PID", "NAME");
}

probe kernel.function("sys_open")
{
    printf("%6d %6d %16s\n", uid(), pid(), execname());
}

probe timer.s(60)
{
    print("Monitoring ends\n")
    exit()
}
```

We would run the following commands:

```
$ source setup-environment wandboard-quad
$ crosstab root@<target_ip> sys_open.stp
```

Yocto does not support running scripts on the target, as that would require building modules on the target, and that is untested.

How it works...

SystemTap scripts are written with its own C/awk like language. They enable us to trace events by instrumenting the kernel code at different locations, such as:

- ▶ Beginning and end of SystemTap sessions
- ▶ Entry, return, or specific offset of kernel and user space functions
- ▶ Timer events
- ▶ Performance hardware counter events

They also enable us to extract data, such as:

- ▶ Thread, process, or user ID
- ▶ Current CPU
- ▶ Process name
- ▶ Time
- ▶ Local variables
- ▶ Kernel and user space backtraces

Additionally, SystemTap also offers the ability to analyze the gathered data, and for different probes to work together. SystemTap includes a wide selection of example scripts and a framework for creating script libraries that can be shared. These tapsets are installed by default and can be extended by the user's own scripts. When a symbol is not defined in a script, SystemTap will search the tapset library for it.

See also

- ▶ The tapset reference at <https://sourceware.org/systemtap/tapsets/>
- ▶ All examples included in the source at <https://sourceware.org/systemtap/examples/>
- ▶ A reference to the systemtap scripting language at <https://sourceware.org/systemtap/langref/>

Using OProfile

OProfile is a statistical profiler released under the GNU GPL license. The version included in the Yocto 1.7 release is a system-wide profiler, which uses the legacy profiling mode with a kernel module to sample hardware performance counters data and a user space daemon to write them to a file. More recent Yocto releases use newer versions that use the performance events subsystem, which we introduced in the *Using the kernel's performance counters* recipe, so they are able to profile processes and workloads as well.

The version included in Yocto 1.7 consists of a kernel module, a user space daemon to collect sample data, and several profiling tools to analyze captured data.

This recipe will focus on the OProfile version included in the 1.7 Yocto release.

Getting ready

To include OProfile in your system, add the following to your `conf/local.conf` file:

```
IMAGE_INSTALL_append += " oprofile"
```

OProfile is also part of the `tools-profile` image feature, so you can also add it with:

```
EXTRA_IMAGE_FEATURES += "tools-profile"
```

OProfile is also included in the `-sdk` images.

OProfile does not need debugging symbols in applications unless annotated results are needed. For callgraph analysis, the binaries must have stack frames information so they should be build with debug optimization by setting the `DEBUG_BUILD` variable in the `conf/local.conf` file:

```
DEBUG_BUILD = "1"
```

To build the kernel driver, configure the Linux kernel with profiling support, `CONFIG_PROFILING`, and the `CONFIG_OPROFILE` configuration variable to build the OProfile module.

OProfile uses the hardware counters support in the SoC, but it can also work on a timer-based mode. To work with the timer-based model, you need to pass the `oprofile.timer=1` kernel argument to the Linux kernel, or load the OProfile module with:

```
# modprobe oprofile timer=1
```



Because OProfile relies on the i.MX6 performance counters, we still need to boot with `maxcpus=1` for it to work. This restricts the profiling in i.MX6 SoCs to one core.

How to do it...

To profile a single ping, start a profiling session as follows:

```
# opcontrol --start --vmlinux=/boot/vmlinux --callgraph 5
Using 2.6+ OProfile kernel interface.
Reading module info.
Using log file /var/lib/oprofile/samples/oprofiled.log
Daemon started.
Profiler running.
```

Then run the workload to profile, for example, a single ping:

```
# ping -c 1 192.168.1.1
PING 192.168.1.1 (192.168.1.1): 56 data bytes
64 bytes from 192.168.1.1: seq=0 ttl=64 time=5.421 ms

--- 192.168.1.1 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 5.421/5.421/5.421 ms
```

And stop collecting data with:

```
# opcontrol --stop
```



We will get a parsing error if the kernel image name contains special characters. To avoid it, we can use a symbolic link as follows:

```
# ln -s /boot/vmlinux-3.10.17-1.0.2-
    wandboard+gbe8d6872b5eb /boot/vmlinux
```

Also, if you see the following error:

```
Count 100000 for event CPU_CYCLES is below the minimum
1500000
```

You will need to change the reset count of the `CPU_CYCLES` event to that minimum, with:

```
# opcontrol --setup --event=CPU_CYCLES:1500000
```

You can then view the collected data with:

```
# opreport -f
Using /var/lib/oprofile/samples/ for samples directory.
CPU: ARM Cortex-A9, speed 996000 MHz (estimated)
Counted CPU_CYCLES events (CPU cycle) with a unit mask of 0x00 (No
unit mask) count 1500000
CPU_CYCLES:150...
samples| %|
-----
401 83.0228 /boot/vmlinux-3.10.17-1.0.2-wandboard+gbe8d6872b5eb
31 6.4182 /bin/bash
28 5.7971 /lib/libc-2.20.so
18 3.7267 /lib/ld-2.20.so
3 0.6211 /usr/bin/oprofiled
1 0.2070 /usr/bin/ophelp
1 0.2070 /usr/sbin/sshd
```

And an excerpt for output with callgraph and symbols is as follows:

```
# opreport -cl
Using /var/lib/oprofile/samples/ for samples directory.
warning: [heap] (tgid:790 range:0x3db000-0x4bc000) could not be
found.
warning: [stack] (tgid:785 range:0x7ee11000-0x7ee32000) could not be
found.
CPU: ARM Cortex-A9, speed 996000 MHz (estimated)
Counted CPU_CYCLES events (CPU cycle) with a unit mask of 0x00 (No
unit mask) count 1500000
samples % app name symbol name
-----
102 48.8038 vmlinux-3.10.17-1.0.2-wandboard+gbe8d6872b5eb
__do_softirq
107 51.1962 vmlinux-3.10.17-1.0.2-wandboard+gbe8d6872b5eb
do_softirq
102 21.1180 vmlinux-3.10.17-1.0.2-wandboard+gbe8d6872b5eb
__do_softirq
102 47.4419 vmlinux-3.10.17-1.0.2-wandboard+gbe8d6872b5eb
__do_softirq
```

```
102      47.4419  vmlinux-3.10.17-1.0.2-wandboard+gbe8d6872b5eb
__do_softirq [self]
7       3.2558  vmlinux-3.10.17-1.0.2-wandboard+gbe8d6872b5eb
net_rx_action
4       1.8605  vmlinux-3.10.17-1.0.2-wandboard+gbe8d6872b5eb
run_timer_softirq
-----
-----
31      6.4182  bash           /bin/bash
```

How it works...

The OProfile daemon records data continuously, accumulating data from multiple runs. Use the `--start` and `--stop` options to start and stop accumulating new data. If you want to start collecting data from scratch, use the `--reset` option first.

Before running a profiling session, you need to configure the OProfile daemon to run with or without kernel profiling. Specifying the kernel profiling option is the only compulsory configuration variable.

In order to configure the OProfile daemon, stop it first (if running) with the `--shutdown` option. The `--stop` option will only stop data collection, but will not kill the daemon.

To configure OProfile without kernel profiling you execute the following command:

```
opcontrol --no-vmlinux <options>
```

And to configure the kernel profiling, we can run the following command:

```
opcontrol --vmlinux=/boot/path/to/vmlinux <options>
```

Both of these will configure the daemon and load the OProfile kernel module, if needed. Some common options are:

- ▶ `--separate=<type>`: This controls how the profiled data is separated into different files, with type being:
 - **none**: This does not separate profiles.
 - **library**: This separates shared libraries profiles per application. The sample file name will include the name of library and the executable.
 - **kernel**: This adds kernel profiling.
 - **thread**: This adds per thread profiles.
 - **cpu**: This adds per CPU profiles.
 - **all**: This does all of the above.

- ▶ `--callgrah=<depth>`: This logs called and calling functions as well as the time spent in functions.

Once the daemon is configured, you can start a profiling session.

To check the current configuration, you execute:

```
# opcontrol --status
Daemon not running
Session-dir: /var/lib/oprofile
Separate options: library kernel
vmlinux file: /boot/vmlinux
Image filter: none
Call-graph depth: 5
```

The sampled data is stored in the `/var/lib/oprofile/samples/` directory.

We can then analyze the collected data with:

```
opreport <options>
```

Some useful options include:

- ▶ `-c`: This shows callgraph information, if available.
- ▶ `-g`: This shows the source file and line number for each symbol.
- ▶ `-f`: This shows full object paths.
- ▶ `-o`: This provides the output to the specified file instead of `stdout`.

OProfile mounts a pseudo filesystem in `/dev/oprofile` which is used to report and receive configuration from user space. It also contains a character device node used to pass sampled data from the kernel module to the user space daemon.

There's more...

Yocto includes a graphical user interface for OProfile that can be run in the host. However, it is not part of Poky and needs to be downloaded and installed separately.

Refer to the `oprofileui` repository at <https://git.yoctoproject.org/cgit.cgi/oprofileui/> for a `README` with instructions, or to the *Yocto Project's Profiling and Tracing Manual* at <http://www.yoctoproject.org/docs/1.7.1/profile-manual/profile-manual.html>.

See also

- ▶ The project's home page for more information about OProfile at <http://oprofile.sourceforge.net/news/>

Using LTTng

LTTng is a set of dual licensed GPLv2 and LGPL tracing and profiling tools for both applications and kernel. It produces binary trace files in the production optimized **Compact Trace Format (CTF)**, which can then be analyzed by tools, such as **babeltrace**.

Getting ready

To include the different LTTng tools in your system, add the following to your `conf/local.conf` file:

```
IMAGE_INSTALL_append = " lttng-tools lttng-modules lttng-ust"
```

They are also part of the `tools-profile` image feature, so you can also add them with:

```
EXTRA_IMAGE_FEATURES += "tools-profile"
```

These are also included in the `-sdk` images.



At the time of writing, Yocto 1.7 excludes `lttng-modules` from the `tools-profile` feature and `sdk` images for ARM; so they have to be added manually.



The LTTng command-line tool is the main user interface to LTTng. It can be used to trace both the Linux kernel—using the kernel tracing interfaces we have seen in previous recipes—as well as instrumented user space applications.

How to do it...

A kernel profiling session workflow is as follows:

1. Create a profiling session with:

```
# lttng create test-session
Session test-session created.
Traces will be written in /home/root/lttng-traces/test-
session-20150117-174945
```

2. Enable the events you want to trace with:

```
# lttng enable-event --kernel sched_switch,sched_process_fork
Warning: No tracing group detected
Kernel event sched_switch created in channel channel0
Kernel event sched_process_fork created in channel channel0
```

You can get a list of the available kernel events with:

```
# lttng list --kernel
```

This corresponds to the static tracepoint events available in the Linux kernel.

3. Now, you are ready to start sampling profiling data:

```
# lttng start
Tracing started for session test-session
```

4. Run the workload you want to profile:

```
# ping -c 1 192.168.1.1
```

5. When the command finishes or is interrupted, stop the gathering of profiling data:

```
# lttng stop
Waiting for data availability.
Tracing stopped for session test-session
```

6. Finally, destroy the profiling session using the following command. Note that this keeps the tracing data and only destroys the session.

```
# lttng destroy
Session test-session destroyed
```

7. To view the profiling data so that it is readable by humans, start babeltrace with:

```
# babeltrace /home/root/lttng-traces/test-session-20150117-
174945
```

The profiling data can also be copied to the host to be analyzed.

User space applications and libraries need to be instrumented so that they can be profiled. This is done by linking them with the liblttng-ust library.

Applications can then make use of the `tracef` function call, which has the same format as `printf()`, to output traces. For example, to instrument the example `helloworld.c` application we saw in previous chapters, modify the source in `meta-custom/recipes-example/helloworld/helloworld-1.0/helloworld.c` as follows:

```
#include <stdio.h>
#include <lttng/tracef.h>

main(void)
{
    printf("Hello World");
    tracef("I said: %s", "Hello World");
}
```

Modify its Yocto recipe in `meta-custom/recipes-example/helloworld/helloworld_1.0.bb` as follows:

```
DESCRIPTION = "Simple helloworld application"
SECTION = "examples"
LICENSE = "MIT"
LIC_FILES_CHKSUM =
    "file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4
     f302"

SRC_URI = "file://helloworld.c"
DEPENDS = "lttng-ust"

S = "${WORKDIR}"

do_compile() {
    ${CC} helloworld.c -lltng-ust -o helloworld
}

do_install() {
    install -d ${D}${bindir}
    install -m 0755 helloworld ${D}${bindir}
}
```

Then build the package, copy it to the target, and start a profiling session as follows:

1. Create a profiling session by executing the following command:

```
# lttng create test-user-session
Session test-user-session created.
Traces will be written in /home/root/lttng-traces/test-user-
session-20150117-185731
```

2. Enable the events you want to profile—in this case, all the user space events:

```
# lttng enable-event -u -a
Warning: No tracing group detected
All UST events are enabled in channel channel0
```

3. Start to gather profiling data:

```
# lttng start
Tracing started for session test-user-session
```

4. Run the workload—in this case, the instrumented hello world example program:

```
# helloworld
Hello World
```

5. Once it finishes, stop gathering data:

```
# lttng stop
Waiting for data availability.
Tracing stopped for session test-user-session
```

6. Without destroying the session, you can start babeltrace executing:

```
# lttng view
[18:58:22.625557512] (+0.001278334) wandboard-quad
lttng_ust_tracef:event: { _cpu_id = 0 }, { _msg_length = 19,
msg = "I said: Hello World" }
```

7. Finally, you can destroy the profiling session:

```
# lttng destroy test-user-session
Session test-user-session destroyed
```

How it works...

Kernel tracing is done using the tracing functionalities available in the Linux kernel, as we have seen in previous recipes. For the following examples to work, the Linux kernel must be configured appropriately as seen in the corresponding recipes earlier.

LTTng provides a common user interface to control some of the kernel tracing features we saw previously, such as the following:

- ▶ **Static tracepoint events:**

You can enable specific static tracepoint events with:

```
# lttng enable-event <comma separated event list> -k
```

You can enable all tracepoints with:

```
# lttng enable-event -a -k --tracepoint
```

You can also enable all syscalls with:

```
# lttng enable-event -a -k --syscall
```

You can enable all tracepoints and syscalls with:

```
# lttng enable-event -a -k
```

- ▶ **Dynamic tracepoint events:**

You can also add dynamic tracepoints with:

```
# lttng enable-event <probe_name> -k --probe <symbol>+<offset>
```

You can also add them with:

```
# lttng enable-event <probe_name> -k --probe <address>
```

- ▶ **Function tracing:**

You can also use the function tracing kernel functionality with:

```
# lttng enable-event <probe_name> -k --function <symbol>
```

- ▶ **Performance counter events:**

And the hardware performance counters, for example for the CPU cycles, with the following command:

```
# lttng add-context -t perf:cpu:cpu-cycles -k
```

Use the `add-context --help` option to list further context options and perf counters.

Extending application profiling

Further applications tracing flexibility can be achieved with the `tracepoint()` call by writing a template file (`.tp`), and using the `lttng-gen-tp` script along with the source file. This generates an object file that can then be linked to your application.

At the time of writing, Yocto has no standard way to cross-instrument user space applications, but it can be done natively using an `-sdk` image, or adding the following image features to the `conf/local.conf` file:

```
EXTRA_IMAGE_FEATURES += " tools-sdk dev-pkgs"
```

For example, define a tracepoint `hw.tp` file as follows:

```
TRACEPOINT_EVENT(
    hello_world_trace_provider,
    hw_tracepoint,
    TP_ARGS(
        int, my_integer_arg,
        char*, my_string_arg
    ),
    TP_FIELDS(
        ctf_string(my_string_field, my_string_arg)
        ctf_integer(int, my_integer_field, my_integer_arg)
    )
)
```

Pass this through the `lttng-gen-tp` tool to obtain `hw.c`, `hw.h`, and `hw.o` files:

```
# lttng-gen-tp hw.tp
```



Note that the `lttng-gen-tp` tool is not installed with the `lttng-ust` package, but with the `lttng-ust-bin` package. This has to be added to be the target image, for example, by adding the following in your `conf/local.conf` file:

```
IMAGE_INSTALL_append = " lttng-ust-bin"
```

You can now add the `hw.h` header file to your helloworld application that is in the `helloworld.c` file and use the `tracepoint()` call as follows:

```
#include <stdio.h>
#include "hw.h"

main(void)
{
    printf("Hello World");
```

```
    tracepoint(hello_world_trace_provider, hw_tracepoint, 1, "I
        said: Hello World");
}
```

Now link your application with the native gcc as follows:

```
# gcc -o hw helloworld.c hw.o -llttng-ust -ldl
```



Note that in order to use gcc on the target, we need to build one of the -sdk images, or add some extra features to our image, such as:

```
EXTRA_IMAGE_FEATURES = "tools-sdk dev-pkgs"
```



To profile your application, do the following:

1. Create a profiling session:

```
# lttng create test-session
Spawning a session daemon
Warning: No tracing group detected
Session test-session created.

Traces will be written in /home/root/lttng-traces/test-
session-20150117-195930
```

2. Enable the specific event you want to profile:

```
# lttng enable-event --userspace
    hello_world_trace_provider:hw_tracepoint
Warning: No tracing group detected
UST event hello_world_trace_provider:hw_tracepoint created in
channel channel0
```

3. Start gathering profiling data:

```
# lttng start
Tracing started for session test-session
```

4. Run the workload to profile—in this case the helloworld application:

```
./hw
Hello World
```

5. Stop gathering data:

```
# lttng stop
```

6. Now start babeltrace with:

```
# lttng view  
[20:00:43.537630037] (+?.?????????) wandboard-quad  
    hello_world_trace_provider:hw_tracepoint: { cpu_id = 0 }, {  
        my_string_field = "I said: Hello World", my_integer_field =  
        1 }
```

7. Finally, destroy the profiling session:

```
# lttng destroy test-session
```

There's more...

You can also use the Trace Compass application or Eclipse plugin to analyze the traces in the host by visiting <http://projects.eclipse.org/projects/tools.tracecompass/downloads>. A stable release was not yet available at the time of writing.

See also

- ▶ Details on using LTTng at <http://lttng.org/docs/>
- ▶ Details about the instrumenting of C applications at <http://lttng.org/docs/#doc-c-application>
- ▶ A tracepoint() example in the lttng-ust source at <http://git.lttng.org/?p=lttng-ust.git;a=tree;f=tests/hello;h=4ae310caf62a8321a253fa84a04982edab52829c;hb=HEAD>

Using blktrace

There are a few tools available to perform block devices I/O monitoring and profiling.

Starting with `iotop` which we mentioned in the *Exploring Yocto's tracing and profiling tools* recipe, which gives a general idea of the throughput on a system and a particular process. Or `iostat`, which provides many more statistics regarding CPU usage and device utilization, but does not provide per process details. And finally `blktrace` that is a GPLv2 licensed tool which monitors specific block devices I/O at a low level, and can also compute **I/O operations per second (IOPS)**.

This recipe will explain how to use `blktrace` to trace block devices and `blkparse`, to convert the traces into human readable format.

Getting ready

To use blktrace and blkparse, you can add them to the target image by adding it specifically, as in:

```
IMAGE_INSTALL_append = " blktrace"
```

Alternately, you can also use the tools-profile image feature, or an -sdk image.

You will also need to configure the Linux kernel with CONFIG_FTRACE and CONFIG_BLK_DEV_IO_TRACE to be able to trace block I/O actions.

When profiling a block device, it is important to minimize the effect of the tracing on the results; for example, not storing the tracing data on the block device being profiled.

There are several ways to achieve this:

- ▶ Running the trace from a different block device.
- ▶ Running the trace from a RAM-based tmpfs device (such as /var/volatile).
Running from a memory-based device will limit the amount of tracing data that can be stored though.
- ▶ Running the trace from a network-mounted filesystem.
- ▶ Running the trace over the network.

Also, the filesystem being used in the block device to profile is an important factor, as filesystem features such as journaling will distort the I/O statistics. Flash filesystems, even if they are presented to user space as block devices, cannot be profiled with blktrace.

How to do it...

Let's imagine you want to profile the I/O for the microSD card device on the Wandboard. By booting the system from the network, as seen in the *Configuring network booting for a development setup* recipe from *Chapter 1, The Build System*, you can avoid unnecessary access to the device by the system.

For this example, we will mount as an ext2 partition to avoid journaling, but other tweaks may be needed for effective profiling of a specific workload:

```
# mount -t ext2 /dev/mmcblk0p2 /mnt
EXT2-fs (mmcblk0p2): warning: mounting ext3 filesystem as ext2
EXT2-fs (mmcblk0p2): warning: mounting unchecked fs, running e2fsck
is recommended
```

The workflow to profile a specific workload is as follows:

1. Start blktrace to gather tracing data on the /dev/mmcblk0 device with:

```
# blktrace /dev/mmcblk0
```
2. Start the workload to profile, for example, the creation of a 10 KB file. Open an SSH connection to the target and execute:

```
# dd if=/dev/urandom of=/mnt/home/root/random-10k-file bs=1k
count=10 conv=fsync
10+0 records in
10+0 records out
10240 bytes (10 kB) copied, 0.00585167 s, 1.7 MB/s
```
3. Stop the profiling on the console with *Ctrl + C*. This will create a file in the same directory called mmcblk0.blktrace.0. You will see the following output:

```
^C== mmcblk0 ==
CPU 0:          30 events,      2 KiB data
Total:          30 events (dropped 0),      2
KiB data
```

Some useful options for blktrace are:

- ▶ -w: This is used to run only for the specified number of seconds
- ▶ -a: This adds a mask to the current file, where the masks can be:
 - barrier: This refers to the barrier attribute
 - complete: This refers to an operation completed by the driver
 - fs: These are the FS requests
 - issue: This option refers to operations issued to the driver
 - pc: This refers to packet command events
 - queue: This option represents queue operations
 - read: This refers to read traces
 - requeue: This is used for requeue operations
 - sync: This represents synchronous attributes
 - write: This refers to write traces

How it works...

Once you have gathered the tracing data, you can process it with blkparse as follows:

```
# blkparse mmcblk0
```

This provides an stdout output for all the gathered data, and a final summary, as follows:

```
Input file mmcblk0.blktrace.0 added
179,0      0      1      0.0000000000  521  A   W 1138688 + 8 <-
(179,2) 1114112
179,0      0      2      0.000003666  521  Q   W 1138688 + 8
[kworker/u8:0]
179,0      0      3      0.000025333  521  G   W 1138688 + 8
[kworker/u8:0]
179,0      0      4      0.000031000  521  P   N [kworker/u8:0]
179,0      0      5      0.000044666  521  I   W 1138688 + 8
[kworker/u8:0]
179,0      0      0      0.000056666  0   m   N cfq519A
insert_request
179,0      0      0      0.000063000  0   m   N cfq519A
add_to_rr
179,0      0      6      0.000081000  521  U   N [kworker/u8:0] 1
179,0      0      0      0.000121000  0   m   N cfq workload
slice:6
179,0      0      0      0.000132666  0   m   N cfq519A
set_active wl_class:0 wl_type:0
179,0      0      0      0.000141333  0   m   N cfq519A Not
idling. st->count:1
179,0      0      0      0.000150000  0   m   N cfq519A fifo=
(null)
179,0      0      0      0.000156000  0   m   N cfq519A
dispatch_insert
179,0      0      0      0.000167666  0   m   N cfq519A
dispatched a request
179,0      0      0      0.000175000  0   m   N cfq519A activate
rq, drv=1
179,0      0      7      0.000181333  83  D   W 1138688 + 8
[mmcqd/2]
179,0      0      8      0.735417000  83  C   W 1138688 + 8 [0]
179,0      0      0      0.739904333  0   m   N cfq519A complete
rqnoidle 0
179,0      0      0      0.739910000  0   m   N cfq519A
set_slice=4
179,0      0      0      0.739912000  0   m   N cfq schedule
CPU0 (mmcblk0):
```

```
Reads Queued:          0,      0KiB  Writes Queued:1,4KiB
Read Dispatches:        0,      0KiB  Write Dispatches:1,4KiB
Reads Requeued:         0                  Writes Requeued:0
Reads Completed:        0,      0KiB  Writes Completed:1,4KiB
Read Merges:             0,      0KiB  Write Merges:0,0KiB
Read depth:              0                  Write depth:1
IO unplugs:             1                  Timer unplugs:0

Throughput (R/W): 0KiB/s / 5KiB/s
Events (mmcblk0): 20 entries
Skips: 0 forward (0 - 0.0%)
```

The output format from blkparse is:

```
179,0      0      7      0.000181333    83  D      W 1138688 + 8
[mmcqd/2]
```

This corresponds to:

```
<mayor,minor> <cpu> <seq_nr> <timestamp> <pid> <actions> <rwb>
<start block> + <nr of blocks> <command>
```

The columns correspond to:

- ▶ A: I/O remapped to a different device
- ▶ B: I/O bounced
- ▶ C: I/O completed
- ▶ D: I/O issued to driver
- ▶ F: I/O front merged with request on queue
- ▶ G: Get request
- ▶ I: I/O inserted into request queue
- ▶ M: I/O back merged with request on queue
- ▶ P: Plug request
- ▶ Q: I/O handled by request queue code
- ▶ S: Sleep request
- ▶ T: Unplug due to timeout
- ▶ U: Unplug request
- ▶ X: Split

The RWBS field corresponds to:

- ▶ R: Read
- ▶ W: Write
- ▶ B: Barrier
- ▶ S: Synchronous

Another way of tracing non-disruptively is using live monitoring, that is, piping the output of blktrace to blkparse directly without writing anything to disk, as follows:

```
# blktrace /dev/mmcblk0 -o - | blkparse -i -
```

This can also be done in just one line:

```
# btrace /dev/mmcblk0
```

There's more...

The blktrace command can also send the tracing data over the network so that it is stored on a different device.

For this, start blktrace on the target system as follows:

```
# blktrace -l /dev/mmcblk0
```

And on another device, run another instance as follows:

```
$ blktrace -d /dev/mmcblk0 -h <target_ip>
```

Back to the target, you can now execute the specific workload you want to trace:

```
# dd if=/dev/urandom of=/mnt/home/root/random-10k-file bs=1k count=10
  conv=fsync
10+0 records in
10+0 records out
10240 bytes (10 kB) copied, 0.00585167 s, 1.7 MB/s
```

Once it finishes, interrupt the remote blktrace with *Ctrl + C*. A summary will be printed at both the target and the host.

You can now run blkparse to process the gathered data.

Module 3

Mastering Embedded Linux Programming

Harness the power of Linux to create versatile and robust embedded solutions

1

Starting Out

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.

- 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 into 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 upstream? 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.

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.

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 BeagleBoard 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 named `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.

2

Learning About Toolchains

The toolchain is the first element of embedded Linux and the starting point of your project. The choices you make at this early stage will have a profound impact on the final outcome. Your toolchain should be capable of making effective use of your hardware by using the optimum instruction set for your processor, using the floating point unit if there is one, and so on. It should support the languages that you require and have a solid implementation of POSIX and other system interfaces. Not only that, but it should be updated when security flaws are discovered or bugs found. Finally, it should be constant throughout the project. In other words, once you have chosen your toolchain it is important to stick with it. Changing compilers and development libraries in an inconsistent way during a project will lead to subtle bugs.

Obtaining a toolchain is as simple as downloading and installing a package. But, the toolchain itself is a complex thing, as I will show you in this chapter.

What is a toolchain?

A toolchain is the set of tools that compiles source code into executables that can run on your target device, and includes a compiler, a linker, and run-time libraries. Initially, you need one to build the other three elements of an embedded Linux system: the bootloader, the kernel, and the root filesystem. It has to be able to compile code written in assembly, C, and C++ since these are the languages used in the base open source packages.

Usually, toolchains for Linux are based on components from the GNU project (<http://www.gnu.org>) and that is still true in the majority of cases at the time of writing. However, over the past few years, the Clang compiler and the associated LLVM project (<http://llvm.org>) have progressed to the point that it is now a viable alternative to a GNU toolchain. One major distinction between LLVM and GNU-based toolchains is in the licensing; LLVM has a BSD license, while GNU has the GPL. There are some technical advantages to Clang as well, such as faster compilation and better diagnostics, but GNU GCC has the advantage of compatibility with the existing code base and support for a wide range of architectures and operating systems. Indeed, there are still some areas where Clang cannot replace the GNU C compiler, especially when it comes to compiling a mainline Linux kernel. It is probable that, in the next year or so, Clang will be able to compile all the components needed for embedded Linux and so will become an alternative to GNU. There is a good description of how to use Clang for cross compilation at <http://clang.llvm.org/docs/CrossCompilation.html>. If you would like to use it as part of an embedded Linux build system, the EmbToolkit (<https://www.embtoolkit.org>) fully supports both GNU and LLVM/Clang toolchains and various people are working on using Clang with Buildroot and the Yocto Project. I will cover embedded build systems in *Chapter 6, Selecting a Build System*. Meanwhile, this chapter focuses on the GNU toolchain as it is the only complete option at this time.

A standard GNU toolchain consists of three main components:

- **Binutils:** A set of binary utilities including the assembler, and the linker, ld. It is available at <http://www.gnu.org/software/binutils/>.
- **GNU Compiler Collection (GCC):** These are the compilers for C and other languages which, depending on the version of GCC, include C++, Objective-C, Objective-C++, Java, Fortran, Ada, and Go. They all use a common back-end which produces assembler code which is fed to the GNU assembler. It is available at <http://gcc.gnu.org/>.
- **C library:** A standardized API based on the POSIX specification which is the principle interface to the operating system kernel from applications. There are several C libraries to consider, see the following section.

As well as these, you will need a copy of the Linux kernel headers, which contain definitions and constants that are needed when accessing the kernel directly. Right now, you need them to be able to compile the C library, but you will also need them later when writing programs or compiling libraries that interact with particular Linux devices, for example to display graphics via the Linux frame buffer driver. This is not simply a question of making a copy of the header files in the include directory of your kernel source code. Those headers are intended for use in the kernel only and contain definitions that will cause conflicts if used in their raw state to compile regular Linux applications.

Instead, you will need to generate a set of sanitized kernel headers which I have illustrated in *Chapter 5, Building a Root Filesystem*.

It is not usually crucial whether the kernel headers are generated from the exact version of Linux you are going to be using or not. Since the kernel interfaces are always backwards-compatible, it is only necessary that the headers are from a kernel that is the same as or older than the one you are using on the target.

Most people would consider the GNU debugger, GDB, to be part of the toolchain as well, and it is usual that it is built at this point. I will talk about GDB in *Chapter 12, Debugging with GDB*.

Types of toolchain - native versus cross toolchain

For our purposes, there are two types of toolchain:

- **Native:** This toolchain runs on the same type of system, sometimes the same actual system, as the programs it generates. This is the usual case for desktops and servers, and it is becoming popular on certain classes of embedded devices. The Raspberry Pi running Debian for ARM, for example, has self-hosted native compilers.
- **Cross:** This toolchain runs on a different type of system than the target, allowing the development to be done on a fast desktop PC and then loaded onto the embedded target for testing.

Almost all embedded Linux development is done using a cross development toolchain, partly because most embedded devices are not well suited to program development since they lack computing power, memory, and storage, but also because it keeps the host and target environments separate. The latter point is especially important when the host and the target are using the same architecture, X86_64, for example. In this case, it is tempting to compile natively on the host and simply copy the binaries to the target. This works up to a point but it is likely that the host distribution will receive updates more often than the target, that different engineers building code for the target will have slightly different versions of the host development libraries and so you will violate the principle that the toolchain should remain constant throughout the life of the project. You can make this approach work if you ensure that the host and target build environments are in lockstep with each other, but a much better approach is to keep the host and the target separate, and a cross toolchain is a way to do that.

However, there is a counter argument in favor of native development. Cross development creates the burden of cross-compiling all the libraries and tools that you need for your target. We will see later on in this chapter that cross-compiling is not always simple because most open source packages are not designed to be built in this way. Integrated build tools, including Buildroot and the Yocto Project, help by encapsulating the rules to cross compile a range of packages that you need in typical embedded systems but, if you want to compile a large number of additional packages, then it is better to natively compile them. For example, to provide a Debian distribution for the Raspberry Pi or BeagleBone using a cross compiler is impossible, they have to be natively compiled. Creating a native build environment from scratch is not easy and involves creating a cross compiler first to bootstrap a native build environment on the target and using that to build packages. You would need a build farm of well-provisioned target boards or you may be able to use QEMU to emulate the target. If you want to look into this further, you may want to look at the Scratchbox project, now in its second incarnation as Scratchbox2 (<https://maemo.gitorious.org/scratchbox2>). It was developed by Nokia to build their Maemo Linux operating system, and is used today by the Mer project and the Tizen project, among others.

Meanwhile, in this chapter, I will focus on the more mainstream cross compiler environment, which is relatively easy to set up and administer.

CPU architectures

The toolchain has to be built according to the capabilities of the target CPU, which includes:

- **CPU architecture:** arm, mips, x86_64, and so on
- **Big- or little-endian operation:** Some CPUs can operate in both modes, but the machine code is different for each
- **Floating point support:** Not all versions of embedded processors implement a hardware floating point unit, in which case, the toolchain can be configured to call a software floating point library instead
- **Application Binary Interface (ABI):** The calling convention used for passing parameters between function calls

With many architectures, the ABI is constant across the family of processors. One notable exception is ARM. The ARM architecture transitioned to the **Extended Application Binary Interface (EABI)** in the late 2000's, resulting in the previous ABI being named the **Old Application Binary Interface (OABI)**. While the OABI is now obsolete, you continue to see references to EABI. Since then, the EABI has split into two, based on the way that floating point parameters are passed. The original EABI uses general purpose (integer) registers while the newer EABIHF uses floating point registers. The EABIHF is significantly faster at floating point operations since it removes the need for copying between integer and floating point registers, but it is not compatible with CPUs that do not have a floating point unit. The choice, then, is between two incompatible ABIs: you cannot mix and match the two and so you have to decide at this stage.

GNU uses a prefix to the tools to identify the various combinations that can be generated, consisting of a tuple of three or four components separated by dashes, as described here:

- **CPU:** The CPU architecture, such as arm, mips, or x86_64. If the CPU has both endian modes, they may be differentiated by adding el for little-endian, or eb for big-endian. Good examples are little-endian MIPS, mipsel and big-endian ARM, armeb.
- **Vendor:** This identifies the provider of the toolchain. Examples include buildroot, poky, or just unknown. Sometimes it is left out altogether.
- **Kernel:** For our purposes, it is always 'linux'.
- **Operating system:** A name for the user space component, which might be gnu or uclibcgnu. The ABI may be appended here as well so, for ARM toolchains, you may see gnueabi, gnueabihf, uclibcgnueabi, or uclibcgnueabihf.

You can find the tuple used when building the toolchain by using the `-dumpmachine` option of `gcc`. For example, you may see the following on the host computer:

```
$ gcc -dumpmachine
x86_64-linux-gnu
```



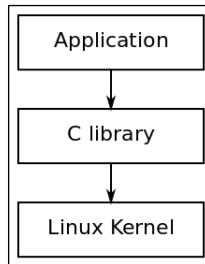
When a native compiler is installed on a machine, it is normal to create links to each of the tools in the toolchain with no prefixes so that you can call the compiler with the command `gcc`.

Here is an example using a cross compiler:

```
$ mipsel-unknown-linux-gnu-gcc -dumpmachine
mipsel-unknown-linux-gnu
```

Choosing the C library

The programming interface to the Unix operating system is defined in the C language, which is now defined by the POSIX standards. The C library is the implementation of that interface; it is the gateway to the kernel for Linux programs, as shown in the following diagram. Even if you are writing programs in another language, maybe Java or Python, the respective run-time support libraries will have to call the C library eventually:



The C library is the gateway to the kernel for applications

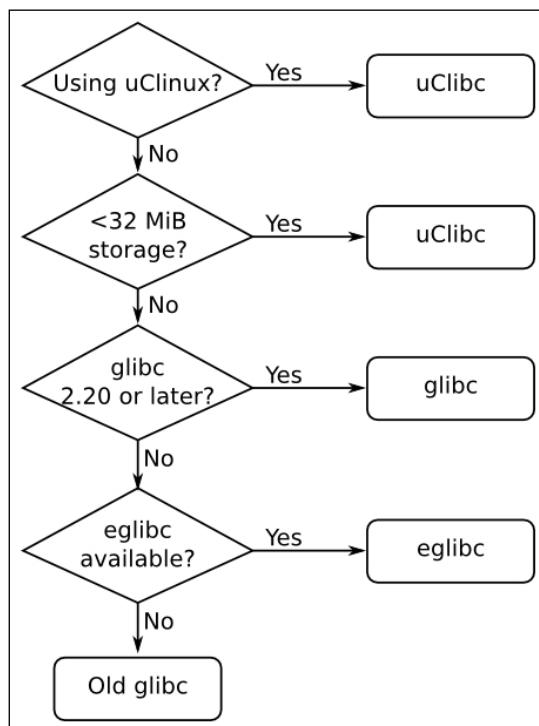
Whenever the C library needs the services of the kernel it will use the kernel system call interface to transition between user space and kernel space. It is possible to bypass the C library by making kernel system calls directly, but that is a lot of trouble and almost never necessary.

There are several C libraries to choose from. The main options are as follows:

- **glibc**: Available at <http://www.gnu.org/software/libc>. It is the standard GNU C library. It is big and, until recently, not very configurable, but it is the most complete implementation of the POSIX API.
- **eglibc**: Available at <http://www.eglibc.org/home>. This is the embedded GLIBC. It was a series of patches to glibc which added configuration options and support for architectures not covered by glibc (specifically, the PowerPC e500). The split between eglibc and glibc was always rather artificial and, fortunately, the code base from eglibc has been merged back into glibc as of version 2.20, leaving us with one improved library. eglibc is no longer maintained.

- **uClibc**: Available at <http://www.uclibc.org>. The 'u' is really a Greek 'mu' character, indicating that this is the micro controller C library. It was first developed to work with uClinux (Linux for CPUs without memory management units), but has since been adapted to be used with full Linux. There is a configuration utility which allows you to fine-tune its features to your needs. Even a full configuration is smaller than glibc but it is not as complete an implementation of the POSIX standards.
- **musl libc**: Available at <http://www.musl-libc.org>. It is a new C library designed for embedded systems.

So, which to choose? My advice is to use uClibc only if you are using uClinux or if you have very limited amounts of storage or RAM and so the small size would be an advantage. Otherwise, I prefer to use an up-to-date glibc, or eglIBC. I have no experience of musl libc but if you find that glibc/eglIBC is not suitable, by all means give it a go. This process is summarized in the following figure:



Choosing a C library

Finding a toolchain

You have three choices for your cross development toolchain: you may find a ready built toolchain that matches your needs, you can use the one generated by an embedded build tool which is covered in *Chapter 6, Selecting a Build System*, or you can create one yourself as described later in this chapter.

A pre-built cross toolchain is an attractive option in that you only have to download and install it, but you are limited to the configuration of that particular toolchain and you are dependent on the person or organization you got it from. Most likely, it will be one of these:

- SoC or board vendor. Most vendors offer a Linux toolchain.
- A consortium dedicated to providing system-level support for a given architecture. For example, Linaro, (<https://www.linaro.org>) have pre-built toolchains for the ARM architecture.
- Third-party Linux tool vendors such as Mentor Graphics, TimeSys, or MontaVista.
- Cross tool packages for your desktop Linux distribution, for example, Debian-based distributions have packages for cross compiling for ARM, MIPS, and PowerPC targets.
- A binary SDK produced by one of the integrated embedded build tools, the Yocto Project has some examples at <http://autobuilder.yoctoproject.org/pub/releases/CURRENT/toolchain> and there is also the Denx Embedded Linux Development Kit at <ftp://ftp.denx.de/pub/eldk/>.
- A link from a forum that you can't find any more.

In all of these cases, you have to decide whether the pre-built toolchain on offer meets your requirements. Does it use the C library you prefer? Will the provider give you updates for security fixes and bugs, bearing in mind my comments on support and updates from *Chapter 1, Starting Out*. If your answer is no to any of these then you should consider creating your own.

Unfortunately, building a toolchain is no easy task. If you truly want to do the whole thing yourself, take a look at *Cross Linux From Scratch* (<http://trac.clfs.org>). There, you will find step-by-step instructions on how to create each component.

A simpler alternative is to use crosstool-NG, which encapsulates the process into a set of scripts and has a menu-driven front-end. You still need a fair degree of knowledge, though, just to make the right choices.

It is simpler still to use a build system such as Buildroot or the Yocto Project since they generate a toolchain as part of the build process. This is my preferred solution as I have shown in *Chapter 6, Selecting a Build System*.

Building a toolchain using crosstool-NG

I am going to begin with crosstool-NG because it allows you to see the process of creating the toolchain and to create several different sorts.

Some years ago, Dan Kegel wrote a set of scripts and makefiles for generating cross development toolchains and called it crosstool (kegel.com/crosstool). In 2007, Yann E. Morin used that base to create the next generation of crosstool, crosstool-NG (crosstool-ng.org). Today it is, by far, the most convenient way to create a stand-alone cross toolchain from source.

Installing crosstool-NG

Before you begin, you will need a working native toolchain and build tools on your host PC. To work with crosstool-NG on an Ubuntu host, you will need to install the packages using the following command:

```
$ sudo apt-get install automake bison chrpath flex g++ git gperf  
gawk libexpat1-dev libncurses5-dev libsdl1.2-dev libtool  
python2.7-dev texinfo
```

Next, get the current release from the croostool-NG downloads section, <http://crosstool-ng.org/download/crosstool-ng>. In my examples I have used 1.20.0. Extract it and create the front-end menu system, ct-ng, as shown in the following commands:

```
$ tar xf crosstool-ng-1.20.0.tar.bz2  
$ cd crosstool-ng-1.20.0  
$ ./configure --enable-local  
$ make  
$ make install
```

The --enable-local option means that the program will be installed into the current directory, which avoids the need for root permissions as would be required if you were to install it in the default location, /usr/local/bin. Type ./ct-ng from the current directory to launch the crosstool menu.

Selecting the toolchain

Crosstool-NG can build many different combinations of toolchains. To make the initial configuration easier, it comes with a set of samples that cover many of the common use-cases. Use `./ct-ng list-samples` to generate the list.

As an example, suppose that your target is a BeagleBone Black which has an ARM Cortex A8 core and a VFPv3 floating point unit, and that you want to use a current version of glibc. The closest sample is `arm-cortex_a8-linux-gnueabi`. You can see the default configuration by prefixing the name with `show`:

```
$ ./ct-ng show-arm-cortex_a8-linux-gnueabi
[L..] arm-cortex_a8-linux-gnueabi
OS           : linux-3.15.4
Companion libs : gmp-5.1.3 mpfr-3.1.2 cloog-ppl-0.18.1 mpc-1.0.2
libelf-0.8.13
binutils      : binutils-2.22
C compiler     : gcc-4.9.1 (C,C++)
C library      : glibc-2.19 (threads: nptl)
Tools          : dmalloc-5.5.2 dumapl-2_5_15 gdb-7.8 ltrace-
0.7.3 strace-4.8
```

To select this as the target configuration, you would type:

```
$ ./ct-ng arm-cortex_a8-linux-gnueabi
```

At this point, you can review the configuration and make changes using the configuration menu command `menuconfig`:

```
$ ./ct-ng menuconfig
```

The menu system is based on the Linux kernel `menuconfig` and so navigation of the user interface will be familiar to anyone who has configured a kernel. If not, please refer to *Chapter 4, Porting and Configuring the Kernel* for a description of `menuconfig`.

There are a few configuration changes that I would recommend you make at this point:

- In **Paths and misc** options, disable **Render the toolchain read-only** (`CT_INSTALL_DIR_RO`)
- In **Target options | Floating point**, select **hardware (FPU)** (`CT_ARCH_FLOAT_HW`)
- In **C-library | extra config**, add **--enable-obsolete-rpc** (`CT_LIBC_GLIBC_EXTRA_CONFIG_ARRAY`)

The first is necessary if you want to add libraries to the toolchain after it has been installed, which I will describe later in this chapter. The next is to select the optimum floating point implementation for a processor with a hardware floating point unit. The last forces the toolchain to be generated with an obsolete header file, `rpc.h`, which is still used by a number of packages (note that this is only a problem if you selected glibc). The names in parentheses are the configuration labels stored in the configuration file. When you have made the changes, exit `menuconfig`, and save the configuration as you do so.

The configuration data is saved into a file named `.config`. Looking at the file, you will see that the first line of text reads *Automatically generated make config: don't edit* which is generally good advice, but I recommend that you ignore it in this case. Do you remember from the discussion about toolchain ABIs that ARM has two variants, one which passes floating point parameters in integer registers and one that uses the VFP registers? The float configuration you have just chosen is of the latter type and so the ABI part of the tuple should read `eabihf`. There is a configuration parameter that does exactly what you want but it is not enabled by default, neither does it appear in the menu, at least not in this version of crosstool. Consequently, you will have to edit `.config` and add the line shown in bold as follows:

```
[...]
#
# arm other options
#
CT_ARCH_ARM_MODE="arm"
CT_ARCH_ARM_MODE_ARM=y
# CT_ARCH_ARM_MODE_THUMB is not set
# CT_ARCH_ARM_INTERWORKING is not set
CT_ARCH_ARM_EABI_FORCE=y
CT_ARCH_ARM_EABI=y
CT_ARCH_ARM_TUPLE_USE_EABIHF=y
[...]
```

Now you can use crosstool-NG to get, configure, and build the components according to your specification by typing the following command:

```
$ ./ct-ng build
```

The build will take about half an hour, after which you will find your toolchain is present in `~/x-tools/arm-cortex_a8-linux-gnueabihf/`.

Anatomy of a toolchain

To get an idea of what is in a typical toolchain, I want to examine the crosstool-NG toolchain you have just created.

The toolchain is in the directory `~/x-tools/arm-cortex_a8-linux-gnueabihf/bin`. In there you will find the cross compiler, `arm-cortex_a8-linux-gnueabihf-gcc`. To make use of it, you need to add the directory to your path using the following command:

```
$ PATH=~/x-tools/arm-cortex_a8-linux-gnueabihf/bin:$PATH
```

Now you can take a simple `Hello World` program that looks like this:

```
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[])
{
    printf ("Hello, world!\n");
    return 0;
}
```

And compile it like this:

```
$ arm-cortex_a8-linux-gnueabihf-gcc helloworld.c -o helloworld
```

You can confirm that it has been cross compiled by using the `file` command to print the type of the file:

```
$ file helloworld
helloworld: ELF 32-bit LSB executable, ARM, version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 3.15.4, not
stripped
```

Finding out about your cross compiler

Imagine that you have just received a toolchain, and that you would like to know more about how it was configured. You can find out a lot by querying `gcc`. For example, to find the version, you use `--version`:

```
$ arm-cortex_a8-linux-gnueabi-gcc --version
arm-cortex_a8-linux-gnueabi-gcc (crosstool-NG 1.20.0) 4.9.1
Copyright (C) 2014 Free Software Foundation, Inc.

This is free software; see the source for copying conditions. There
is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR
PURPOSE.
```

To find how it was configured, use -v:

```
$ arm-cortex_a8-linux-gnueabi-gcc -v
Using built-in specs.
COLLECT_GCC=arm-cortex_a8-linux-gnueabihf-gcc
COLLECT_LTO_WRAPPER=/home/chris/x-tools/arm-cortex_a8-linux-
gnueabihf/libexec/gcc/arm-cortex_a8-linux-gnueabihf/4.9.1/lto-wrapper
Target: arm-cortex_a8-linux-gnueabihf
Configured with:
/home/chris/hd/home/chris/build/MELP/build/crosstool-ng-
1.20.0/.build/src/gcc-4.9.1/configure --build=x86_64-build_unknown-
linux-gnu --host=x86_64-build_unknown-linux-gnu --target=arm-
cortex_a8-linux-gnueabihf --prefix=/home/chris/x-tools/arm-cortex_a8-
linux-gnueabihf/arm-cortex_a8-linux-gnueabihf/sysroot --enable-
languages=c,c++ --with-arch=armv7-a --with-cpu=cortex-a8 --with-
tune=cortex-a8 --with-float=hard --with-pkgversion='crosstool-NG
1.20.0' --enable-__cxa_atexit --disable-libmudflap --disable-libgomp
--disable-libssp --disable-libquadmath --disable-libquadmath-support
--disable-lbsanitizer --with-
gmp=/home/chris/hd/home/chris/build/MELP/build/crosstool-ng-
1.20.0/.build/arm-cortex_a8-linux-gnueabihf/buildtools --with-
mpfr=/home/chris/hd/home/chris/build/MELP/build/crosstool-ng-
1.20.0/.build/arm-cortex_a8-linux-gnueabihf/buildtools --with-
mpc=/home/chris/hd/home/chris/build/MELP/build/crosstool-ng-
1.20.0/.build/arm-cortex_a8-linux-gnueabihf/buildtools --with-
isl=/home/chris/hd/home/chris/build/MELP/build/crosstool-ng-
1.20.0/.build/arm-cortex_a8-linux-gnueabihf/buildtools --with-
cloog=/home/chris/hd/home/chris/build/MELP/build/crosstool-ng-
1.20.0/.build/arm-cortex_a8-linux-gnueabihf/buildtools --with-
libelf=/home/chris/hd/home/chris/build/MELP/build/crosstool-ng-
1.20.0/.build/arm-cortex_a8-linux-gnueabihf/buildtools --with-host-
libstdcxx=' -static-libgcc -Wl,-Bstatic,-lstdc++,-Bdynamic -lm' --
enable-threads=posix --enable-target-optspace --enable-plugin --
enable-gold --disable-nls --disable-multilib --with-local-
prefix=/home/chris/x-tools/arm-cortex_a8-linux-gnueabihf/arm-
cortex_a8-linux-gnueabihf/sysroot --enable-c99 --enable-long-long
Thread model: posix
gcc version 4.9.1 (crosstool-NG 1.20.0)
```

There is a lot of output there but the interesting things to note are:

- `--with-sysroot=/home/chris/x-tools/arm-cortex_a8-linux-
gnueabihf/arm-cortex_a8-linux-gnueabihf/sysroot`: This is the
default sysroot directory, see the following section for an explanation
- `--enable-languages=c,c++`: Using this we have both C and C++
languages enabled

- `--with-arch=armv7-a`: The code is generated using the ARM v7a instruction set
- `--with-cpu=cortex-a8` and `--with-tune=cortex-a8`: The code is further tweaked for a Cortex A8 core
- `--with-float=hard`: Generate opcodes for the floating point unit and uses the VFP registers for parameters
- `--enable-threads=posix`: Enable POSIX threads

These are the default settings for the compiler. You can override most of them on the gcc command line so, for example, if you want to compile for a different CPU, you can override the configured setting, `--with-cpu`, by adding `-mcpu` to the command line, as follows:

```
$ arm-cortex_a8-linux-gnueabihf-gcc -mcpu=cortex-a5 helloworld.c -o helloworld
```

You can print out the range of architecture-specific options available using `--target-help`, as follows:

```
$ arm-cortex_a8-linux-gnueabihf-gcc --target-help
```

You may be wondering if it matters whether or not you get the exact configuration right at the time you generate the toolchain if you can change it later on, and the answer depends on the way you anticipate using it. If you plan to create a new toolchain for each target, then it makes sense to set everything up at the beginning because it will reduce the risks of getting it wrong later on. Jumping ahead a little to *Chapter 6, Selecting a Build System*, I call this the Buildroot philosophy. If, on the other hand, you want to build a toolchain that is generic and you are prepared to provide the correct settings when you build for a particular target, then you should make the base toolchain generic, which is the way the Yocto Project handles things.

The preceding examples follow the Buildroot philosophy.

The sysroot, library, and header files

The toolchain sysroot is a directory which contains subdirectories for libraries, header files, and other configuration files. It can be set when the toolchain is configured through `--with-sysroot` or it can be set on the command line, using `--sysroot`. You can see the location of the default sysroot by using `-print-sysroot`:

```
$ arm-cortex_a8-linux-gnueabi-gcc -print-sysroot
/home/chris/x-tools/arm-cortex_a8-linux-gnueabihf/arm-cortex_a8-
linux-gnueabihf/sysroot
```

You will find the following in the sysroot:

- `lib`: Contains the shared objects for the C library and the dynamic linker/loader, `ld-linux`
- `usr/lib`: the static library archives for the C library and any other libraries that may be installed subsequently
- `usr/include`: Contains the headers for all the libraries
- `usr/bin`: Contains the utility programs that run on the target, such as the `ldd` command
- `/usr/share`: Used for localization and internationalization
- `sbin`: Provides the `ldconfig` utility, used to optimize library loading paths

Plainly, some of these are needed on the development host to compile programs, and others – for example the shared libraries and `ld-linux` – are needed on the target at runtime.

Other tools in the toolchain

The following table shows various other commands in the toolchain together with a brief description:

Command	Description
<code>addr2line</code>	Converts program addresses into filenames and numbers by reading the debug symbol tables in an executable file. It is very useful when decoding addresses printed out in a system crash report.
<code>ar</code>	The archive utility is used to create static libraries.
<code>as</code>	This is the GNU assembler.
<code>c++filt</code>	This is used to demangle C++ and Java symbols.
<code>cpp</code>	This is the C preprocessor, and is used to expand <code>#define</code> , <code>#include</code> , and other similar directives. You seldom need to use this by itself.
<code>elfedit</code>	This is used to update the ELF header of ELF files.
<code>g++</code>	This is the GNU C++ front-end, which assumes source files contain C++ code.
<code>gcc</code>	This is the GNU C front-end, which assumes source files contain C code.

Command	Description
gcov	This is a code coverage tool.
gdb	This is the GNU debugger.
gprof	This is a program profiling tool.
ld	This is the GNU linker.
nm	This lists symbols from object files.
objcopy	This is used to copy and translate object files.
objdump	This is used to display information from object files.
ranlib	This creates or modifies an index in a static library, making the linking stage faster.
readelf	This displays information about files in ELF object format.
size	This lists section sizes and the total size.
strings	This display strings of printable characters in files.
strip	This is used to strip an object file of debug symbol tables, thus making it smaller. Typically, you would strip all the executable code that is put onto the target.

Looking at the components of the C library

The C library is not a single library file. It is composed of four main parts that together implement the POSIX functions API:

- `libc`: The main C library that contains the well-known POSIX functions such as `printf`, `open`, `close`, `read`, `write`, and so on
- `libm`: Maths functions such as `cos`, `exp`, and `log`
- `libpthread`: All the POSIX thread functions with names beginning with `pthread_`
- `librt`: The real-time extensions to POSIX, including shared memory and asynchronous I/O

The first one, `libc`, is always linked in but the others have to be explicitly linked with the `-l` option. The parameter to `-l` is the library name with `lib` stripped off. So, for example, a program that calculates a sine function by calling `sin()` would be linked with `libm` using `-lm`:

```
arm-cortex_a8-linux-gnueabihf-gcc myprog.c -o myprog -lm
```

You can verify which libraries have been linked in this or any other program by using the `readelf` command:

```
$ arm-cortex_a8-linux-gnueabihf-readelf -a myprog | grep "Shared library"
0x00000001 (NEEDED)           Shared library: [libm.so.6]
0x00000001 (NEEDED)           Shared library: [libc.so.6]
```

Shared libraries need a run-time linker, which you can expose using:

```
$ arm-cortex_a8-linux-gnueabihf-readelf -a myprog | grep "program interpreter"
[Requesting program interpreter: /lib/ld-linux-armhf.so.3]
```

This is so useful that I have a script file with these commands into a shell script:

```
#!/bin/sh
${CROSS_COMPILE}readelf -a $1 | grep "program interpreter"
${CROSS_COMPILE}readelf -a $1 | grep "Shared library"
```

Linking with libraries: static and dynamic linking

Any application you write for Linux, whether it be in C or C++, will be linked with the C library, `libc`. This is so fundamental that you don't even have to tell `gcc` or `g++` to do it because it always links `libc`. Other libraries that you may want to link with have to be explicitly named through the `-l` option.

The library code can be linked in two different ways: statically, meaning that all the library functions your application calls and their dependencies are pulled from the library archive and bound into your executable; and dynamically, meaning that references to the library files and functions in those files are generated in the code but the actual linking is done dynamically at runtime.

Static libraries

Static linking is useful in a few circumstances. For example, if you are building a small system which consists of only BusyBox and some script files, it is simpler to link BusyBox statically and avoid having to copy the runtime library files and linker. It will also be smaller because you only link in the code that your application uses rather than supplying the entire C library. Static linking is also useful if you need to run a program before the filesystem that holds the runtime libraries is available.

You tell gcc to link all libraries statically by adding `-static` to the command line:

```
$ arm-cortex_a8-linux-gnueabihf-gcc -static helloworld.c -o  
helloworld-static
```

You will notice that the size of the binary increases dramatically:

```
$ ls -l  
-rwxrwxr-x 1 chris chris 5323 Oct  9 09:01 helloworld  
-rwxrwxr-x 1 chris chris 625704 Oct  9 09:01 helloworld-static
```

Static linking pulls code from a library archive, usually named `lib[name].a`. In the preceding case it is `libc.a`, which is in `[sysroot]/usr/lib`:

```
$ ls -l $(arm-cortex_a8-linux-gnueabihf-gcc -print-  
sysroot)/usr/lib/libc.a  
-r----- 1 chris chris 3434778 Oct  8 14:00 /home/chris/x-  
tools/arm-cortex_a8-linux-gnueabihf/arm-cortex_a8-linux-  
gnueabihf/sysroot/usr/lib/libc.a
```

Note that the syntax `$(arm-cortex_a8-linux-gnueabihf-gcc -print-sysroot)` places the output of the program on the command line. I am using it as a generic way to refer to the files in the sysroot.

Creating a static library is as simple as creating an archive of object files using the `ar` command. If I had two source files named `test1.c` and `test2.c` and I want to create a static library named `libtest.a`, then I would do this:

```
$ arm-cortex_a8-linux-gnueabihf-gcc -c test1.c  
$ arm-cortex_a8-linux-gnueabihf-gcc -c test2.c  
$ arm-cortex_a8-linux-gnueabihf-ar rc libtest.a test1.o test2.o  
$ ls -l  
total 24  
-rw-rw-r-- 1 chris chris 2392 Oct  9 09:28 libtest.a  
-rw-rw-r-- 1 chris chris 116 Oct  9 09:26 test1.c
```

```
-rw-rw-r-- 1 chris chris 1080 Oct  9 09:27 test1.o
-rw-rw-r-- 1 chris chris   21 Oct  9 09:26 test2.c
-rw-rw-r-- 1 chris chris 1088 Oct  9 09:27 test2.o
```

Then I could link libtest into my helloworld program using:

```
$ arm-cortex_a8-linux-gnueabihf-gcc helloworld.c -ltest -L../libs -
I../libs -o helloworld
```

Shared libraries

A more common way to deploy libraries is as shared objects that are linked at runtime, which makes more efficient use of storage and system memory, since only one copy of the code needs to be loaded. It also makes it easy to update library files without having to re-link all the programs that use them.

The object code for a shared library must be position-independent so that the runtime linker is free to locate it in memory at the next free address. To do this, add the `-fPIC` parameter to gcc, and then link it using the `-shared` option:

```
$ arm-cortex_a8-linux-gnueabihf-gcc -fPIC -c test1.c
$ arm-cortex_a8-linux-gnueabihf-gcc -fPIC -c test2.c
$ arm-cortex_a8-linux-gnueabihf-gcc -shared -o libtest.so test1.o
test2.o
```

To link an application with this library, you add `-ltest`, exactly as in the static case mentioned in the preceding paragraph but, this time, the code is not included in the executable, but there is a reference to the library that the runtime linker will have to resolve:

```
$ arm-cortex_a8-linux-gnueabihf-gcc helloworld.c -ltest -L../libs -
I../libs -o helloworld
$ list-libs helloworld
[Requesting program interpreter: /lib/ld-
linux-armhf.so.3]
0x00000001 (NEEDED)                               Shared library: [libtest.so]
0x00000001 (NEEDED)                               Shared library: [libc.so.6]
```

The runtime linker for this program is `/lib/ld-linux-armhf.so.3`, which must be present in the target's filesystem. The linker will look for `libtest.so` in the default search path: `/lib` and `/usr/lib`. If you want it to look for libraries in other directories as well, you can place a colon-separated list of paths in the shell variable `LD_LIBRARY_PATH`:

```
# export LD_LIBRARY_PATH=/opt/lib:/opt/usr/lib
```

Understanding shared library version numbers

One of the benefits of shared libraries is that they can be updated independently of the programs that use them. Library updates are of two types: those that fix bugs or add new functions in a backwards-compatible way, and those that break compatibility with existing applications. GNU/Linux has a versioning scheme to handle both these cases.

Each library has a release version and an interface number. The release version is simply a string that is appended to the library name, for example the JPEG image library, libjpeg, is currently at release 8.0.2 and so the library is named `libjpeg.so.8.0.2`. There is a symbolic link named `libjpeg.so` to `libjpeg.so.8.0.2` so that, when you compile a program with `-ljpeg`, you link with the current version. If you install version 8.0.3, the link is updated and you will link with that one instead.

Now, suppose that version 9.0.0 comes along and that breaks backwards compatibility. The link from `libjpeg.so` now points to `libjpeg.so.9.0.0`, so that any new programs are linked with the new version, possibly throwing compile errors when the interface to libjpeg changes, which the developer can fix. Any programs on the target that are not recompiled are going to fail in some way because they are still using the old interface. This is where the soname helps. The soname encodes the interface number when the library was built and is used by the runtime linker when it loads the library. It is formatted as `<library name>.so.<interface number>`. For `libjpeg.so.8.0.2`, the soname is `libjpeg.so.8`:

```
$ readelf -a /usr/lib/libjpeg.so.8.0.2 | grep SONAME
0x000000000000000e (SONAME)           Library soname:
[libjpeg.so.8]
```

Any program compiled with it will request `libjpeg.so.8` at runtime which will be a symbolic link on the target to `libjpeg.so.8.0.2`. When version 9.0.0 of libjpeg is installed, it will have a soname of `libjpeg.so.9`, and so it is possible to have two incompatible versions of the same library installed on the same system. Programs that were linked with `libjpeg.so.8.*.*` will load `libjpeg.so.8`, and those linked with `libjpeg.so.9.*.*` will load `libjpeg.so.9`.

This is why, when you look at the directory listing of `<sysroot>/usr/lib/libjpeg*`, you find these four files:

- `libjpeg.a`: This is the library archive used for static linking
- `libjpeg.so -> libjpeg.so.8.0.2` : This is a symbolic link, used for dynamic linking

- `libjpeg.so.8 -> libjpeg.so.8.0.2`: This is a symbolic link used when loading the library at runtime
- `libjpeg.so.8.0.2`: This is the actual shared library, used at both compile time and runtime

The first two are only needed on the host computer for building, the last two are needed on the target at runtime.

The art of cross compiling

Having a working cross toolchain is the starting point of a journey, not the end of it. At some point, you will want to begin cross compiling the various tools, applications, and libraries that you need on your target. Many of them will be open source packages, each of which has its own method of compiling, and each with its own peculiarities. There are some common build systems, including:

- Pure makefiles where the toolchain is controlled by the `make` variable `CROSS_COMPILE`
- The GNU build system known as Autotools
- CMake (<https://cmake.org>)

I will cover only the first two here since these are the ones needed for even a basic embedded Linux system. For CMake, there are some excellent resources on the CMake website referenced in the preceding point.

Simple makefiles

Some important packages are very simple to cross compile, including the Linux kernel, the U-Boot bootloader, and Busybox. For each of these, you only need to put the toolchain prefix in the `make` variable `CROSS_COMPILE`, for example `arm-cortex_a8-linux-gnueabi-`. Note the trailing dash `-`.

So, to compile Busybox, you would type:

```
$ make CROSS_COMPILE=arm-cortex_a8-linux-gnueabi-
```

Or, you can set it as a shell variable:

```
$ export CROSS_COMPILE=arm-cortex_a8-linux-gnueabi-
$ make
```

In the case of U-Boot and Linux, you also have to set the `make` variable `ARCH` to one of the machine architectures they support, which I will cover in *Chapter 3, All About Bootloaders* and *Chapter 4, Porting and Configuring the Kernel*.

Autotools

The name, Autotools, refers to a group of tools that are used as the build system in many open source projects. The components, together with the appropriate project pages, are:

- GNU Autoconf
(<http://www.gnu.org/software/autoconf/autoconf.html>)
- GNU Automake
(<http://www.gnu.org/savannah-checkouts/gnu/automake>)
- GNU Libtool (<http://www.gnu.org/software/libtool/libtool.html>)
- Gnulib (<https://www.gnu.org/software/gnulib>)

The role of Autotools is to smooth over the differences between the many different types of system that the package may be compiled for, accounting for different versions of compilers, different versions of libraries, different locations of header files, and dependencies with other packages. Packages that use Autotools come with a script named `configure` that checks dependencies and generates makefiles according to what it finds. The `configure` script may also give you the opportunity to enable or disable certain features. You can find the options on offer by running `./configure --help`.

To configure, build, and install a package for the native operating system, you would typically run these three commands:

```
$ ./configure  
$ make  
$ sudo make install
```

Autotools is able to handle cross development as well. You can influence the behavior of the `configure` script by setting these shell variables:

- CC: The C compiler command
- CFLAGS: Additional C compiler flags
- LDFLAGS: Additional linker flags, for example if you have libraries in a non-standard directory `<lib dir>` you would add it to the library search path by adding `-L<lib dir>`
- LIBS: Contains a list of additional libraries to pass to the linker, for instance `-lm` for the math library

- **CPPFLAGS**: Contains C/C++ preprocessor flags, for example you would add `-I<include dir>` to search for headers in a non-standard directory `<include dir>`
- **CPP**: The C preprocessor to use

Sometimes it is sufficient to set only the `cc` variable, as follows:

```
$ CC=arm-cortex_a8-linux-gnueabihf-gcc ./configure
```

At other times, that will result in an error like this:

```
[...]
checking whether we are cross compiling... configure: error: in '/home/
chris/MELP/build/sqlite-autoconf-3081101':
configure: error: cannot run C compiled programs.
If you meant to cross compile, use '--host'.
See 'config.log' for more details
```

The reason for the failure is that `configure` often tries to discover the capabilities of the toolchain by compiling snippets of code and running them to see what happens, which cannot work if the program has been cross compiled. Nevertheless, there is a hint in the error message of how to solve the problem. Autotools understands three different types of machine that may be involved when compiling a package:

- **Build**: This is the computer that is to build the package, which defaults to the current machine.
- **Host**: This is the computer the program will run on: for a native compile this is left blank and it defaults to be the same computer as build. For a cross compile you set it to be the tuple of your toolchain.
- **Target**: This is the computer the program will generate code for: you would set this when building a cross compiler, for example.

So, to cross compile, you just need to override host, as follows:

```
$ CC=arm-cortex_a8-linux-gnueabihf-gcc \
./configure --host=arm-cortex_a8-linux-gnueabihf
```

One final thing to note is that the default install directory is `<sysroot>/usr/local/*`. You would usually install it in `<sysroot>/usr/*` so that the header files and libraries would be picked up from their default locations. The complete command to configure a typical Autotools package is:

```
$ CC=arm-cortex_a8-linux-gnueabihf-gcc \
./configure --host=arm-cortex_a8-linux-gnueabihf --prefix=/usr
```

An example: SQLite

The SQLite library implements a simple relational database and is quite popular on embedded devices. You begin by getting a copy of SQLite:

```
$ wget http://www.sqlite.org/2015/sqlite-autoconf-3081101.tar.gz  
$ tar xf sqlite-autoconf-3081101.tar.gz  
$ cd sqlite-autoconf-3081101
```

Next, run the configure script:

```
$ CC=arm-cortex_a8-linux-gnueabihf-gcc \  
.configure --host=arm-cortex_a8-linux-gnueabihf --prefix=/usr
```

That seems to work! If it failed, there would be error messages printed to the terminal and recorded in `config.log`. Note that several makefiles have been created, so now you can build it:

```
$ make
```

Finally, you install it into the toolchain directory by setting the `make` variable `DESTDIR`. If you don't, it will try to install it into the host computer's `/usr` directory which is not what you want.

```
$ make DESTDIR=$(arm-cortex_a8-linux-gnueabihf-gcc -print-sysroot)  
install
```

You may find that final command fails with a file permissions error. A crosstool-NG toolchain will be read-only by default, which is why it is useful to set `CT_INSTALL_DIR_RO` to `y` when building it. Another common problem is that the toolchain is installed in a system directory such as `/opt` or `/usr/local` in which case you will need root permissions when running the install.

After installing, you should find that various files have been added to your toolchain:

- `<sysroot>/usr/bin`: `sqlite3`. This is a command-line interface for SQLite that you can install and run on the target.
- `<sysroot>/usr/lib`: `libssqlite3.so.0.8.6`, `libssqlite3.so.0`, `libssqlite3.so` `libssqlite3.la` `libssqlite3.a`. These are the shared and static libraries.
- `<sysroot>/usr/lib/pkgconfig`: `sqlite3.pc`: This is the package configuration file, as described in the following section.
- `<sysroot>/usr/lib/include`: `sqlite3.h`, `sqlite3ext.h`: These are the header files.
- `<sysroot>/usr/share/man/man1`: `sqlite3.1`. This is the manual page.

Now you can compile programs that use sqlite3 by adding `-lssqlite3` at the link stage:

```
$ arm-cortex_a8-linux-gnueabihf-gcc -lssqlite3 sqlite-test.c -o
sqlite-test
```

Where, `sqlite-test.c` is a hypothetical program that calls SQLite functions. Since `sqlite3` has been installed into the sysroot, the compiler will find the header and library files without any problem. If they had been installed elsewhere you would have to add `-L<lib dir>` and `-I<include dir>`.

Naturally, there will be runtime dependencies as well, and you will have to install the appropriate files into the target directory as described in *Chapter 5, Building a Root Filesystem*.

Package configuration

Tracking package dependencies is quite complex. The package configuration utility, `pkg-config` (<http://www.freedesktop.org/wiki/Software/pkg-config>) helps track which packages are installed and which compile flags each needs by keeping a database of Autotools packages in `[sysroot]/usr/lib/pkgconfig`. For instance, the one for SQLite3 is named `sqlite3.pc` and contains essential information needed by other packages that need to make use of it:

```
$ cat $(arm-cortex_a8-linux-gnueabihf-gcc -print-
sysroot)/usr/lib/pkgconfig/sqlite3.pc
# Package Information for pkg-config
prefix=/usr
exec_prefix=${prefix}
libdir=${exec_prefix}/lib
includedir=${prefix}/include
Name: SQLite
Description: SQL database engine
Version: 3.8.11.1
Libs: -L${libdir} -lssqlite3
Libs.private: -ldl -lpthread
Cflags: -I${includedir}
```

You can use the utility `pkg-config` to extract information in a form that you can feed straight to `gcc`. In the case of a library like `libssqlite3`, you want to know the library name (`--libs`) and any special C flags (`--cflags`):

```
$ pkg-config sqlite3 --libs --cflags
Package sqlite3 was not found in the pkg-config search path.
Perhaps you should add the directory containing `sqlite3.pc'
to the PKG_CONFIG_PATH environment variable
No package 'sqlite3' found
```

Oops! That failed because it was looking in the host's sysroot and the development package for `libssqlite3` has not been installed on the host. You need to point it at the sysroot of the target toolchain by setting the shell variable `PKG_CONFIG_LIBDIR`:

```
$ PKG_CONFIG_LIBDIR=$(arm-cortex_a8-linux-gnueabihf-gcc -print-
sysroot)/usr/lib/pkgconfig \
pkg-config sqlite3 --libs --cflags
-lsqlite3
```

Now the output is `-lsqlite3`. In this case, you knew that already, but generally you wouldn't, so this is a valuable technique. The final command to compile would be:

```
$ PKG_CONFIG_LIBDIR=$(arm-cortex_a8-linux-gnueabihf-gcc -print-
sysroot)/usr/lib/pkgconfig \
arm-cortex_a8-linux-gnueabihf-gcc $(pkg-config sqlite3 --cflags --
libs) sqlite-test.c -o sqlite-
```

Problems with cross compiling

`SqLite3` is a well-behaved package and cross compiles nicely but not all packages are so tame. Typical pain points include:

- Home-grown build systems, `zlib`, for example, has a `configure` script but it does not behave like the Autotools `configure` described in the previous section
- `Configure` scripts that read `pkg-config` information, headers, and other files from the host, disregarding the `--host` override
- Scripts that insist on trying to run cross compiled code

Each case requires careful analysis of the error and additional parameters to the configure script to provide the correct information or patches to the code to avoid the problem altogether. Bear in mind that one package may have many dependencies, especially with programs that have a graphical interface using GTK or QT or handle multimedia content. As an example, mplayer, which is a popular tool for playing multimedia content, has dependencies on over 100 libraries. It would take weeks of effort to build them all.

Therefore, I would not recommend manually cross compiling components for the target in this way except when there is no alternative, or the number of packages to build is small. A much better approach is to use a build tool such as Buildroot or the Yocto Project, or, avoid the problem altogether by setting up a native build environment for your target architecture. Now you can see why distributions like Debian are always compiled natively.

Summary

The toolchain is always your starting point: everything that follows from that is dependent on having a working, reliable toolchain.

Most embedded build environments are based on a cross development toolchain which creates a clear separation between a powerful host computer building the code and a target computer on which it runs. The toolchain itself consists of the GNU binutils, a C compiler from the GNU compiler collection – and quite likely the C++ compiler as well – plus one of the C libraries I have described. Usually the GNU debugger, gdb, will be generated at this point, which I describe in *Chapter 12, Debugging with GDB*. Also, keep a watch out for the Clang compiler, as it will develop over the next few years.

You may start with nothing but a toolchain – perhaps built using crosstool-NG or downloaded from Linaro – and use it to compile all of the packages that you need on your target, accepting the amount of hard work this will entail. Or, you may obtain the toolchain as part of a distribution which includes a range of packages. A distribution can be generated from source code using a build system such as Buildroot or the Yocto Project, or it can be a binary distribution from a third party, maybe a commercial enterprise like Mentor Graphics or an open source project such as the Denx ELDK. Beware of toolchains or distributions that are offered to you for free as part of a hardware package: they are often poorly configured and not maintained. In any case, you should make the choice according to your situation, and then be consistent in its use throughout the project.

Once you have a toolchain, you can use it to build the other components of your embedded Linux system. In the next chapter, you will learn about the bootloader, which brings your device to life and begins the boot process.

3

All About Bootloaders

The bootloader is the second element of embedded Linux. It is the part that starts the system up and loads the operating system kernel. In this chapter, I will look at the role of the bootloader and, in particular, how it passes control from itself to the kernel using a data structure called a device tree, also known as a **flattened device tree** or **FDT**. I will cover the basics of device trees so that you will be able to follow the connections described in a device tree and relate it to real hardware.

I will look at the popular open source bootloader U-Boot and see how to use it to boot a target device and also how to customize it to a new device. Finally, I will take a quick look at Barebox, a bootloader that shares its past with U-Boot but which has, arguably, a cleaner design.

What does a bootloader do?

In an embedded Linux system, the bootloader has two main jobs: basic system initialization and the loading of the kernel. In fact, the first job is somewhat subsidiary to the second in that it is only necessary to get as much of the system working as is needed to load the kernel.

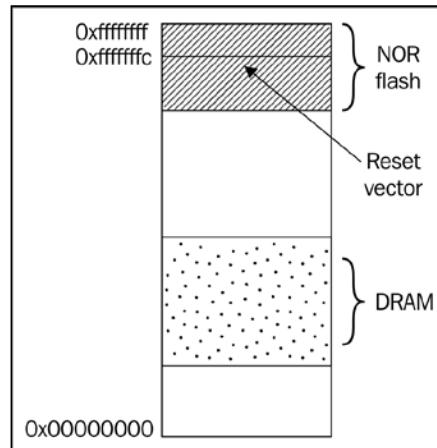
When the first lines of bootloader code are executed, following power-on or a reset, the system is in a very minimal state. The DRAM controller will not have been set up so main memory is not accessible, likewise other interfaces will not have been configured so storage accessed via NAND flash controllers, MMC controllers, and so on, are also not usable. Typically, the only resources operational at the beginning are a single CPU core and some on-chip static memory. As a result, system bootstrap consists of several phases of code, each bringing more of the system into operation.

The early boot phase stops once the interfaces required to load a kernel are working. That includes main memory and the peripherals used to access the kernel and other images, be they mass storage or network. The final act of the bootloader is to load the kernel into RAM and create an execution environment for it. The details of the interface between the bootloader and the kernel are architecture-specific but, in all cases, it means passing a pointer to information about the hardware that the bootloader knows about and passing a kernel command line, which is an ASCII string containing essential information for Linux. Once the kernel has begun executing, the bootloader is no longer needed and all the memory it was using can be reclaimed.

A subsidiary job of the bootloader is to provide a maintenance mode for updating boot configurations, loading new boot images into memory and, maybe, running diagnostics. This is usually controlled by a simple command-line user interface, commonly over a serial interface.

The boot sequence

In simpler times, some years ago, it was only necessary to place the bootloader in non-volatile memory at the reset vector of the processor. NOR flash memory was common at that time and, since it can be mapped directly into the address space, it was the ideal method of storage. The following diagram shows such a configuration, with the reset vector at 0xfffffffffc at the top end of an area of flash memory. The bootloader is linked so that there is a jump instruction at that location that points to the start of the bootloader code:



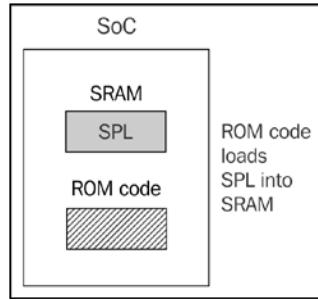
Boot in the old days

From that point, it can initialize the memory controller so that the main memory, the DRAM, becomes available and copies itself into DRAM. Once fully operational, the bootloader can load the kernel from flash memory into DRAM and transfer control to it.

However, once you move away from a simple linearly addressable storage medium like NOR flash, the boot sequence becomes a complex, multi-stage procedure. The details are very specific to each SoC, but they generally follow each of the following phases.

Phase 1: ROM code

In the absence of reliable external memory, the code that runs immediately after a reset or power-on has to be stored on-chip in the SoC; this is known as ROM code. It is programmed into the chip when it is manufactured, hence ROM code is proprietary and cannot be replaced by an open source equivalent. The ROM code can make very few assumptions about any hardware that is not on the chip, because it will be different from one design to another. This applies even to the DRAM chips used for the main system memory. Consequently, the only RAM that the ROM code has access to is the small amount of static RAM (SRAM) found in most SoC designs. The size of the SRAM varies from as little as 4 KiB up to a few hundred KiB:



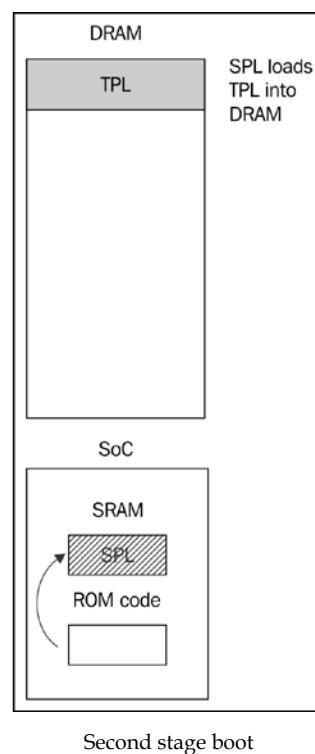
Phase 1 bootloader

The ROM code is capable of loading a small chunk of code from one of several preprogrammed locations into the SRAM. As an example, TI OMAP and Sitara chips will try to load code from the first few pages of NAND flash memory, or from flash memory connected through **SPI (Serial Peripheral Interface)**, or from the first sectors of an MMC device (which could be an eMMC chip or an SD card), or from a file named `MLO` on the first partition of an MMC device. If reading from all of those memory devices fails, then it will try reading a byte stream from Ethernet, USB, or UART; the latter is provided mainly as a means of loading code into flash memory during production rather than for use in normal operation. Most embedded SoCs have ROM code that works in a similar way. In SoCs where the SRAM is not large enough to load a full bootloader like U-Boot, there has to be an intermediate loader called the secondary program loader, or SPL.

At the end of this phase, the next stage bootloader is present in on-chip memory and the ROM code jumps to the beginning of that code.

Phase 2: SPL

The SPL must set up the memory controller and other essential parts of the system preparatory to loading the **third stage program loader** (TPL) into main memory, the DRAM. The functionality of the SPL is limited by its size. It can read a program from a list of storage devices, as can the ROM code, once again using preprogrammed offsets from the start of a flash device, or a well known file name such as `u-boot.bin`. The SPL usually doesn't allow for any user interaction but it may print version information and progress messages which you will see on the console. The following diagram explains the phase 2 architecture:

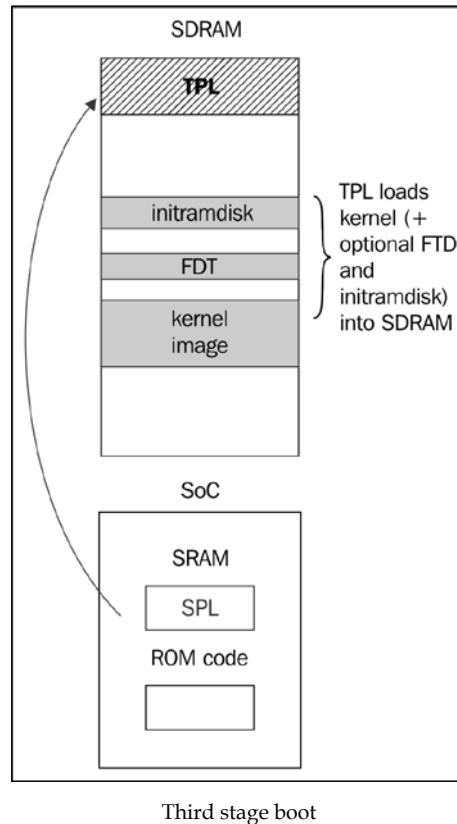


The SPL may be open source, as is the case with the TI x-loader and Atmel AT91Bootstrap, but it is quite common for it to contain proprietary code that is supplied by the manufacturer as a binary blob.

At the end of the second phase, the third stage loader is present in DRAM, and the SPL can make a jump to that area.

Phase 3: TPL

Now, at last, we are running a full bootloader like U-Boot or Barebox. Usually, there is a simple command-line user interface that will let you perform maintenance tasks such as loading new boot and kernel images into flash storage, loading and booting a kernel, and there is a way to load the kernel automatically without user intervention. The following diagram explains the phase 3 architecture:



At the end of the third phase, there is a kernel in memory, waiting to be started. Embedded bootloaders usually disappear from memory once the kernel is running and perform no further part in the operation of the system.

Booting with UEFI firmware

Most embedded PC designs and some ARM designs have firmware based on the **Universal Extensible Firmware Interface (UEFI)** standard, see the official website at <http://www.uefi.org> for more information. The boot sequence is fundamentally the same as described in the preceding section:

Phase 1: The processor loads the UEFI boot manager firmware from flash memory. In some designs, it is loaded directly from NOR flash memory, in others there is ROM code on-chip which loads the boot manager from SPI flash memory. The boot manager is roughly equivalent to the SPL, but may allow user interaction through a text-based or graphical interface.

Phase 2: The boot manager loads the boot firmware from the **EFI System Partition (ESP)** or a hard disk or SSD, or from a network server via PXE boot. If loading from a local disk drive, the ESP is identified by a well-known GUID value of C12A7328-F81F-11D2-BA4B-00A0C93EC93B. The partition should be formatted using the FAT32 format. The third stage bootloader should be in a file named <efi_system_partition>/boot/boot<machine_type_short_name>.efi.

For example, the file path to the loader on an x86_64 system is: /efi/boot/bootx64.efi

Phase 3: The TPL in this case has to be a bootloader that is capable of loading a Linux kernel and an optional RAM disk into memory. Common choices are:

- **GRUB 2:** This is the GNU Grand Unified Bootloader, version 2, and it is the most commonly used Linux loader on PC platforms. However, there is one controversy in that it is licensed under GPL v3, which may make it incompatible with secure booting since the license requires the boot keys to be supplied with the code. The website is <https://www.gnu.org/software/grub/>.
- **gummiboot:** This is a simple UEFI-compatible bootloader which has since been integrated into systemd, and is licensed under LGPL v2.1. The website is <https://wiki.archlinux.org/index.php/Systemd-boot>.

Moving from bootloader to kernel

When the bootloader passes control to the kernel it has to pass some basic information to the kernel, which may include some of the following:

- On PowerPC and ARM architectures: a number unique to the type of the SoC
- Basic details of the hardware detected so far, including at least the size and location of the physical RAM, and the CPU clock speed

- The kernel command line
- Optionally, the location and size of a device tree binary
- Optionally, the location and size of an initial RAM disk

The kernel command line is a plain ASCII string which controls the behavior of Linux, setting, for example, the device that contains the root filesystem. I will look at the details of this in the next chapter. It is common to provide the root filesystem as a RAM disk, in which case it is the responsibility of the bootloader to load the RAM disk image into memory. I will cover the way you create initial RAM disks in *Chapter 5, Building a Root Filesystem*.

The way this information is passed is dependent on the architecture and has changed in recent years. For instance, with PowerPC, the bootloader simply used to pass a pointer to a board information structure, whereas, with ARM, it passed a pointer to a list of "A tags". There is a good description of the format of the kernel source in Documentation/arm/Booting.

In both cases, the amount of information passed was very limited, leaving the bulk of it to be discovered at runtime or hard-coded into the kernel as "platform data". The widespread use of platform data meant that each device had to have a kernel configured and modified for that platform. A better way was needed, and that way is the device tree. In the ARM world, the move away from A tags began in earnest in February 2013 with the release of Linux 3.8 but there are still quite a lot of devices in the field, and even in development, still using A tags.

Introducing device trees

You are almost certainly going to encounter device trees at some point. This section aims to give you a quick overview of what they are and how they work but there are many details that are not discussed.

A device tree is a flexible way to define the hardware components of a computer system. Usually, the device tree is loaded by the bootloader and passed to the kernel, although it is possible to bundle the device tree with the kernel image itself to cater for bootloaders that are not capable of handling them separately.

The format is derived from a Sun Microsystems bootloader known as OpenBoot, which was formalized as the Open Firmware specification, IEEE standard IEEE1275-1994. It was used in PowerPC-based Macintosh computers and so was a logical choice for the PowerPC Linux port. Since then, it has been adapted on a large scale by the many ARM Linux implementations and, to a lesser extent, by MIPS, MicroBlaze, ARC, and other architectures.

I would recommend visiting <http://devicetree.org> for more information.

Device tree basics

The Linux kernel contains a large number of device tree source files in `arch/$ARCH/boot/dts`, and this is a good starting point for learning about device trees. There are also a smaller number of sources in the U-boot source code in `arch/$ARCH/dts`. If you acquired your hardware from a third party, the `dts` file forms part of a board support package and you should expect to receive one along with the other source files.

The device tree represents a computer system as a collection of components joined together in a hierarchy, like a tree. The device tree begins with a root node, represented by a forward slash, `/`, which contains subsequent nodes representing the hardware of the system. Each node has a name and contains a number of properties in the form `name = "value"`. Here is a simple example:

```
/dts-v1/;
/{
    model = "TI AM335x BeagleBone";
    compatible = "ti,am33xx";
    #address-cells = <1>;
    #size-cells = <1>;
    cpus {
        #address-cells = <1>;
        #size-cells = <0>;
        cpu@0 {
            compatible = "arm,cortex-a8";
            device_type = "cpu";
            reg = <0>;
        };
    };
    memory@0x80000000 {
        device_type = "memory";
        reg = <0x80000000 0x20000000>; /* 512 MB */
    };
}
```

Here we have a root node which contains a `cpus` node and a `memory` node. The `cpus` node contains a single CPU node named `cpu@0`. It is a common convention that the names of nodes include an `@` followed by an address that distinguishes them from any others.

Both the root and CPU nodes have a compatible property. The Linux kernel uses this to match this name against the strings exported by device drivers in a struct `of_device_id` (more on this in *Chapter 8, Introducing Device Drivers*). It is a convention that the value is composed of a manufacturer name and a component name to reduce confusion between similar devices made by different manufacturers, hence `ti,am33xx` and `arm,cortex-a8`. It is also quite common to have more than one value for `compatible` where there is more than one driver that can handle this device. They are listed with the most suitable first.

The CPU node and the memory node have a `device_type` property which describes the class of device. The node name is often derived from the `device_type`.

The reg property

The memory and CPU nodes have a `reg` property, which refers to a range of units in a register space. A `reg` property consists of two values representing the start address and the size (length) of the range. Both are written down as zero or more 32-bit integers, called cells. Hence, the memory node refers to a single bank of memory that begins at `0x80000000` and is `0x20000000` bytes long.

Understanding `reg` properties becomes more complex when the address or size values cannot be represented in 32 bits. For example, on a device with 64-bit addressing, you need two cells for each:

```
/ {
    #address-cells = <2>;
    #size-cells = <2>;
    memory@80000000 {
        device_type = "memory";
        reg = <0x00000000 0x80000000 0 0x80000000>;
    };
}
```

The information about the number of cells required is held in `#address-cells` and `#size-cells` declarations in an ancestor node. In other words, to understand a `reg` property, you have to look backwards down the node hierarchy until you find `#address-cells` and `#size-cells`. If there are none, the default values are 1 for each – but it is bad practice for device tree writers to depend on fall-backs.

Now, let's return to the `cpu` and `cpus` nodes. CPUs have addresses as well: in a quad core device they might be addressed as 0, 1, 2, and 3. That can be thought of as a one-dimensional array without any depth so the size is zero. Therefore, you can see that we have `#address-cells = <1>` and `#size-cells = <0>` in the `cpus` node, and in the child node, `cpu@0`, we assign a single value to the `reg` property: `node reg = <0>`.

Phandles and interrupts

The structure of the device tree described so far assumes that there is a single hierarchy of components, whereas in fact there are several. As well as the obvious data connection between a component and other parts of the system, it might also be connected to an interrupt controller, to a clock source and to a voltage regulator. To express these connections, we have phandles.

Take an example of a system containing a serial port which can generate interrupts and the interrupt controller:

```
/dts-v1/;
{
    intc: interrupt-controller@48200000 {
        compatible = "ti,am33xx-intc";
        interrupt-controller;
        #interrupt-cells = <1>;
        reg = <0x48200000 0x1000>;
    };
    serial@44e09000 {
        compatible = "ti,omap3-uart";
        ti,hwmods = "uart1";
        clock-frequency = <48000000>;
        reg = <0x44e09000 0x2000>;
        interrupt-parent = <&intc>;
        interrupts = <72>;
    };
};
```

We have an interrupt-controller node which has the special property `#interrupt-cells`, which tells us how many 4-byte values are needed to represent an interrupt line. In this case, it is just one giving the IRQ number, but it is quite common to use additional values to characterize the interrupt, for example 1 = low-to-high edge triggered, 2 = high-to-low edge triggered, and so on.

Looking at the serial node, it has an `interrupt-parent` property which references the interrupt-controller it is connected to by using its label. This is the phandle. The actual IRQ line is given by the `interrupts` property, 72 in this case.

The serial node has other properties that we have not seen before: `clock-frequency` and `ti,hwmods`. These are part of the bindings for this particular type of device, in other words, the kernel device driver will read these properties to manage the device. The bindings can be found in the Linux kernel source, in directory `Documentation/devicetree/bindings/`.

Device tree include files

A lot of hardware is common between SoCs of the same family and between boards using the same SoC. This is reflected in the device tree by splitting out common sections into include files, usually with the extension `.dtsi`. The Open Firmware standard defines `/include/` as the mechanism to be used, as in this snippet from `vexpress-v2p-ca9.dts`:

```
/include/ "vexpress-v2m.dtsi"
```

Look through the `.dts` files in the kernel, though, and you will find an alternative `include` statement that is borrowed from C, for example in `am335x-boneblack.dts`:

```
#include "am33xx.dtsi"
#include "am335x-bone-common.dtsi"
```

Here is another example from `am33xx.dtsi`:

```
#include <dt-bindings/gpio/gpio.h>
#include <dt-bindings/pinctrl/am33xx.h>
```

Lastly, `include/dt-bindings/pinctrl/am33xx.h` contains normal C macros:

```
#define PULL_DISABLE (1 << 3)
#define INPUT_EN (1 << 5)
#define SLEWCTRL_SLOW (1 << 6)
#define SLEWCTRL_FAST 0
```

All of this is resolved if the device tree sources are built using kernel kbuild, which first runs them through the C pre-processor, `cpp`, where the `#include` and `#define` statements are processed into plain text that is suitable for the device tree compiler. The motivation is shown in the previous example: it means that device tree sources can use the same constant definitions as the kernel code.

When we include files in this way the nodes are overlaid on top of one another to create a composite tree in which the outer layers extend or modify the inner ones. For example, `am33xx.dtsi`, which is general to all am33xx SoCs, defines the first MMC controller interface like this:

```
mmc1: mmc@48060000 {
    compatible = "ti,omap4-hsmmc";
    ti,hwmods = "mmc1";
    ti,dual-volt;
    ti,needs-special-reset;
    ti,needs-special_hs-handling;
    dmas = <&edma 24  &edma 25>;
```

```
dma-names = "tx", "rx";
interrupts = <64>;
interrupt-parent = <&intc>;
reg = <0x48060000 0x1000>;
status = "disabled";
};
```



Note that the status is disabled, meaning that no device driver should be bound to it, and also that it has the label mmc1.



In `am335x-bone-common.dtsi`, which is included with both BeagleBone and BeagleBone Black, the same node is referenced by its phandle:

```
&mmc1 {
    status = "okay";
    bus-width = <0x4>;
    pinctrl-names = "default";
    pinctrl-0 = <&mmc1_pins>;
    cd-gpios = <&gpio0 6 GPIO_ACTIVE_HIGH>;
    cd-inverted;
};
```

Here, `mmc1` is enabled (`status="okay"`) because both variants have a physical MMC1 device, and the `pinctrl` is established. Then, in `am335x-boneblack.dts`, you will see another reference to `mmc1` which associates it with a voltage regulator:

```
&mmc1 {
    vmmc-supply = <&vmmcsd_fixed>;
};
```

So, layering source files like this gives flexibility and reduces the need for duplicated code.

Compiling a device tree

The bootloader and kernel require a binary representation of the device tree, so it has to be compiled using the device tree compiler, `dtc`. The result is a file ending with `.dtb`, which is referred to as a device tree binary or a device tree blob.

There is a copy of `dtc` in the Linux source, in `scripts/dtc/dtc`, and it is also available as a package on many Linux distributions. You can use it to compile a simple device tree (one that does not use `#include`) like this:

```
$ dtc simpledts-1.dts -o simpledts-1.dtb
DTC: dts->dts on file "simpledts-1.dts"
```

Be wary of the fact that `dtc` does not give helpful error messages and it makes no checks other than on the basic syntax of the language, which means that debugging a typing error in a source file can be a lengthy business.

To build more complex examples, you will have to use the kernel `kbuild`, as shown in the next chapter.

Choosing a bootloader

Bootloaders come in all shapes and sizes. The kind of characteristics you want from a bootloader are that they be simple and customizable with lots of sample configurations for common development boards and devices. The following table shows a number of them that are in general use:

Name	Architectures
Das U-Boot	ARM, Blackfin, MIPS, PowerPC, SH
Barebox	ARM, Blackfin, MIPS, PowerPC
GRUB 2	X86, X86_64
RedBoot	ARM, MIPS, PowerPC, SH
CFE	Broadcom MIPS
YAMON	MIPS

We are going to focus on U-Boot because it supports a good number of processor architectures and a large number of individual boards and devices. It has been around for a long time and has a good community for support.

It may be that you received a bootloader along with your SoC or board. As always, take a good look at what you have and ask questions about where you can get the source code from, what the update policy is, how they will support you if you want to make changes, and so on. You may want to consider abandoning the vendor-supplied loader and use the current version of an open source bootloader instead.

U-Boot

U-Boot, or to give its full name, Das U-Boot, began life as an open source bootloader for embedded PowerPC boards. Then, it was ported to ARM-based boards and later to other architectures, including MIPS, SH, and x86. It is hosted and maintained by Denx Software Engineering. There is plenty of information available, and a good place to start is www.denx.de/wiki/U-Boot. There is also a mailing list at u-boot@lists.denx.de.

Building U-Boot

Begin by getting the source code. As with most projects, the recommended way is to clone the git archive and check out the tag you intend to use which, in this case, is the version that was current at the time of writing:

```
$ git clone git://git.denx.de/u-boot.git  
$ cd u-boot  
$ git checkout v2015.07
```

Alternatively, you can get a tarball from <ftp://ftp.denx.de/pub/u-boot/>.

There are more than 1,000 configuration files for common development boards and devices in the `configs/` directory. In most cases, you can make a good guess of which to use, based on the filename, but you can get more detailed information by looking through the per-board README files in the `board/` directory, or you can find information in an appropriate web tutorial or forum. Beware, though, the way U-Boot is configured has undergone a lot of changes since the 2014.10 release. Double-check that the instructions you are following are appropriate.

Taking the BeagleBone Black as an example, we find that there is a likely configuration file named `am335x_boneblack_defconfig` in `configs/` and we find the text **The binary produced by this board supports ... Beaglebone Black** in the board README files for the am335x chip, `board/ti/am335x/README`. With this knowledge, building U-Boot for a BeagleBone Black is simple. You need to inform U-Boot of the prefix for your cross compiler by setting the `make` variable `CROSS_COMPILE` and then select the configuration file using a command of the type `make [board]_defconfig`, as follows:

```
$ make CROSS_COMPILE=arm-cortex-a8-linux-gnueabihf-  
am335x_boneblack_defconfig  
$ make CROSS_COMPILE=arm-cortex-a8-linux-gnueabihf-
```

The results of the compilation are:

- `u-boot`: This is U-Boot in ELF object format, suitable for use with a debugger
- `u-boot.map`: This is the symbol table
- `u-boot.bin`: This is U-Boot in raw binary format, suitable for running on your device
- `u-boot.img`: This is `u-boot.bin` with a U-Boot header added, suitable for uploading to a running copy of U-Boot
- `u-boot.srec`: This is U-Boot in Motorola srec format, suitable for transferring over a serial connection

The BeagleBone Black also requires a **Secondary Program Loader (SPL)**, as described earlier. This is built at the same time and is named MLO:

```
$ ls -l MLO u-boot*
-rw-rw-r-- 1 chris chris 76100 Dec 20 11:22 MLO
-rwxrwxr-x 1 chris chris 2548778 Dec 20 11:22 u-boot
-rw-rw-r-- 1 chris chris 449104 Dec 20 11:22 u-boot.bin
-rw-rw-r-- 1 chris chris 449168 Dec 20 11:22 u-boot.img
-rw-rw-r-- 1 chris chris 434276 Dec 20 11:22 u-boot.map
-rw-rw-r-- 1 chris chris 1347442 Dec 20 11:22 u-boot.srec
```

The procedure is similar for other targets.

Installing U-Boot

Installing a bootloader on a board for the first time requires some outside assistance. If the board has a hardware debug interface, such as JTAG, it is usually possible to load a copy of U-Boot directly into RAM and set it running. From that point, you can use U-Boot commands to copy it into flash memory. The details of this are very board-specific and outside the scope of this book.

Some SoC designs have a boot ROM built in which can be used to read boot code from various external sources such as SD cards, serial interfaces, or USBs, and this is the case with the AM335x chip in the BeagleBone Black. Here is how to load U-Boot via the micro-SD card.

Firstly, format a micro-SD card so that the first partition is in FAT32 format, and mark it as bootable. If you have a direct SD slot available, the card appears as /dev/mmcblk0, otherwise, if you are using a memory card reader, it will be seen as /dev/sdb, or /dev/sdc, and so on. Now, type the following command to partition the micro-SD card, assuming that the card is seen as /dev/mmcblk0:

```
$ sudo sfdisk -D -H 255 -S 63 /dev/mmcblk0 << EOF
,9,0x0C,*
...
EOF
```

Format the first partition as FAT16:

```
$ sudo mkfs.vfat -F 16 -n boot /dev/mmcblk0p1
```

Now, mount the partition you have just formatted: on some systems it is enough to simply remove the micro-SD card and then plug it back in again, on others you may have to click on an icon. On current versions of Ubuntu, it should be mounted as `/media/[user]/boot` so I would copy U-Boot and the SPL to it like this:

```
cp MLO u-boot.img /media/chris/boot
```

Finally, unmount it.

With no power on the BeagleBone board, insert the micro-SD card.

Plug in the serial cable. A serial port should appear on your PC as `/dev/ttyUSB0` or similar.

Start a suitable terminal program such as `gtkterm`, `minicom`, or `picocom` and attach to the port at 115,200 bps with no flow control:

```
$ gtkterm -p /dev/ttyUSB0 -s 115200
```

Press and hold the **Boot Switch** button on the Beaglebone, power up the board using the external 5V power connector, and release the button after about 5 seconds. You should see a U-Boot prompt on the serial console:

```
U-Boot#
```

Using U-Boot

In this section, I will describe some of the common tasks that you can use U-Boot to perform.

Usually, U-Boot offers a command-line interface over a serial port. It gives a command prompt which is customized for each board. In the examples, I will use `U-Boot#`.

Typing `help` prints out all the commands configured in this version of U-Boot; typing `help <command>` prints out more information about a particular command.

The default command interpreter is quite simple. There is no command-line editing by pressing cursor left or right keys; there is no command completion by pressing the `Tab` key; there is no command history by pressing the cursor up key. Pressing any of these keys will disrupt the command you are currently trying to type and you will have to type `Ctrl+C` and start over again. The only line editing key you can safely use is the back space. As an option, you can configure a different command shell called Hush, which has more sophisticated interactive support.

The default number format is hexadecimal. For example, as shown in this command:

```
nand read 82000000 400000 200000
```

This command will read 0x200000 bytes from offset 0x400000 from the start of the NAND flash memory into RAM address 0x82000000.

Environment variables

U-Boot uses environment variables extensively to store and pass information between functions and even to create scripts. Environment variables are simple name=value pairs that are stored in an area of memory. The initial population of variables may be coded in the board configuration header file, like this:

```
#define CONFIG_EXTRA_ENV_SETTINGS \
"myvar1=value1\0" \
"myvar2=value2\0"
```

You can create and modify variables from the U-Boot command line using `setenv`. For example `setenv foo bar` creates the variable `foo` with the value `bar`. Note that there is no `=` sign between the variable name and the value. You can delete a variable by setting it to a null string, `setenv foo`. You can print all the variables to the console using `printenv`, or a single variable using `printenv foo`.

Usually, it is possible to use the `saveenv` command to save the entire environment to permanent storage of some kind. If there is raw NAND or NOR flash, then an erase block is reserved for this purpose, often with another used for a redundant copy to guard against corruption. If there is eMMC or SD card storage it can be stored in a file in a partition of the disk. Other options include storing in a serial EEPROM connected via an I2C or SPI interface or non-volatile RAM.

Boot image format

U-Boot doesn't have a filesystem. Instead, it tags blocks of information with a 64-byte header so that it can track the contents. You prepare files for U-Boot using the `mkimage` command. Here is a brief summary of its usage:

```
$ mkimage
Usage: mkimage -l image
-l ==> list image header information
mkimage [-x] -A arch -O os -T type -C comp -a addr -e ep -n name -d
data_file[:data_file...] image
-A ==> set architecture to 'arch'
```

```
-O ==> set operating system to 'os'  
-T ==> set image type to 'type'  
-C ==> set compression type 'comp'  
-a ==> set load address to 'addr' (hex)  
-e ==> set entry point to 'ep' (hex)  
-n ==> set image name to 'name'  
-d ==> use image data from 'datafile'  
-x ==> set XIP (execute in place)  
mkimage [-D dtc_options] -f fit-image.its fit-image  
mkimage -V ==> print version information and exit
```

For example, to prepare a kernel image for an ARM processor, the command is:

```
$ mkimage -A arm -O linux -T kernel -C gzip -a 0x80008000 \  
-e 0x80008000 -n 'Linux' -d zImage uImage
```

Loading images

Usually, you will load images from removable storage such as an SD card or a network. SD cards are handled in U-Boot by the `mmc` driver. A typical sequence to load an image into memory would be:

```
U-Boot# mmc rescan  
U-Boot# fatload mmc 0:1 82000000 uimage  
reading uimage  
4605000 bytes read in 254 ms (17.3 MiB/s)  
U-Boot# iminfo 82000000  
  
## Checking Image at 82000000 ...  
Legacy image found  
Image Name: Linux-3.18.0  
Created: 2014-12-23 21:08:07 UTC  
Image Type: ARM Linux Kernel Image (uncompressed)  
Data Size: 4604936 Bytes = 4.4 MiB  
Load Address: 80008000  
Entry Point: 80008000  
Verifying Checksum ... OK
```

The command `mmc rescan` re-initializes the `mmc` driver, perhaps to detect that an SD card has recently been inserted. Next, `fatload` is used to read a file from a FAT-formatted partition on the SD card. The format is:

```
fatload <interface> [<dev[:part]> [<addr> [<filename> [bytes  
[pos]]]]]
```

If `<interface>` is mmc, as in our case, `<dev:part>` is the device number of the mmc interface counting from zero, and the partition number counting from one. Hence `<0:1>` is the first partition on the first device. The memory location, `0x82000000`, is chosen to be in an area of RAM that is not being used at this moment. If we intend to boot this kernel, we have to make sure that this area of RAM will not be overwritten when the kernel image is decompressed and located at the runtime location, `0x80008000`.

To load image files over a network you use the **Trivial File Transfer Protocol (TFTP)**. This requires you to install a TFTP daemon, `tftpd`, on your development system and start it running. You also have to configure any firewalls between your PC and the target board to allow the TFTP protocol on UDP port 69 to pass through. The default configuration of `tftpd` allows access only to the directory `/var/lib/tftpboot`. The next step is to copy the files you want to transfer to the target into that directory. Then, assuming that you are using a pair of static IP addresses, which removes the need for further network administration, the sequence of commands to load a set of kernel image files should look like this:

Finally, let's look at how to program images into NAND flash memory and read them back, which is handled by the nand command. This example loads a kernel image via TFTP and programs it into flash:

```
U-Boot# fatload mmc 0:1 82000000 uimage
reading uimage
4605000 bytes read in 254 ms (17.3 MiB/s)

U-Boot# nandecc hw
U-Boot# nand erase 280000 400000

NAND erase: device 0 offset 0x280000, size 0x400000
Erasing at 0x660000 -- 100% complete.
OK

U-Boot# nand write 82000000 280000 400000

NAND write: device 0 offset 0x280000, size 0x400000
4194304 bytes written: OK
```

Now you can load the kernel from flash memory using nand read:

```
U-Boot# nand read 82000000 280000 400000
```

Booting Linux

The bootm command starts a kernel image running. The syntax is:

```
bootm [address of kernel] [address of ramdisk] [address of dtb].
```

The address of the kernel image is necessary, but the address of ramdisk and dtb can be omitted if the kernel configuration does not need them. If there is a dtb but no ramdisk, the second address can be replaced with a dash (-). That would look like this:

```
U-Boot# bootm 82000000 - 83000000
```

Automating the boot with U-Boot scripts

Plainly, typing a long series of commands to boot your board each time it is turned on is not acceptable. To automate the process, U-Boot stores a sequence of commands in environment variables. If the special variable named `bootcmd` contains a script, it is run at power-up after a delay of `bootdelay` seconds. If you are watching this on the serial console, you will see the delay counting down to zero. You can press any key during this period to terminate the countdown and enter into an interactive session with U-Boot.

The way that you create scripts is simple, though not easy to read. You simply append commands separated by semicolons, which must be preceded by a backslash escape character. So, for example, to load a kernel image from an offset in flash memory and boot it, you might use the following command:

```
setenv bootcmd nand read 82000000 400000 200000\;bootm 82000000
```

Porting U-Boot to a new board

Let's assume that your hardware department has created a new board called "Nova" that is based on the BeagleBone Black and that you need to port U-Boot to it. You will need to understand the layout of the U-Boot code and how the board configuration mechanism works. In the 2014.10 release, U-Boot adopted the same configuration mechanism as the Linux kernel, `Kconfig`. Over the next few releases, the existing configuration settings will be moved from the current location in the header files in `include/configs` into `Kconfig` files. As of the 2014.10 release, each board had a `Kconfig` file which contains minimal information derived from the old `boards.cfg` file.

The main directories you will be dealing with are:

- `arch`: Contains code specific to each supported architecture in directories `arm`, `mips`, `powerpc`, and so on. Within each architecture, there is a subdirectory for each member of the family, for example, in `arch/arm/cpu`, there are directories for the architecture variants, including `amt926ejs`, `armv7`, and `armv8`.
- `board`: Contains code specific to a board. Where there are several boards from the same vendor, they can be collected together into a subdirectory, hence the support for the `am335x evm` board, on which the BeagleBone is based, is in `board/ti/am335x`.
- `common`: Contains core functions including the command shells and the commands that can be called from them, each in a file named `cmd_[command name].c`.

- `doc`: Contains several `README` files describing various aspects of U-Boot. If you are wondering how to proceed with your U-Boot port, this is a good place to start.
- `include`: In addition to many shared header files, this contains the very important subdirectory `include/configs` where you will find the majority of the board configuration settings. As the move to Kconfig progresses, the information will be moved out into Kconfig files but, at the time of writing, that process has only just begun.

Kconfig and U-Boot

The way that Kconfig extracts configuration information from Kconfig files and stores the total system configuration in a file named `.config` is described in some detail in *Chapter 4, Porting and Configuring the Kernel*. U-Boot has adopted kconfig and kbuild with one change. A U-Boot build can produce up to three binaries: a normal `u-boot.bin`, a **Secondary Program Loader (SPL)**, and a **Tertiary Program Loader (TPL)**, each with possibly different configuration options. Consequently, lines in `.config` and default configuration files can be prefixed with the codes shown in the following table to indicate which target they apply to:

None	Normal image only
S:	SPL image only
T:	TPL image only
ST:	SPL and TPL images
+S:	Normal and SPL images
+T:	Normal and TPL images
+ST:	Normal, SPL and TPL images

Each board has a default configuration stored in `configs/ [board name]_defconfig`. For your Nova board, you will have to create a file named `nova_defonfig`. for example, and add these lines to it:

```
CONFIG_SPL=y
CONFIG_SYS_EXTRA_OPTIONS="SERIAL1,CONS_INDEX=1,EMMC_BOOT"
+S:CONFIG_ARM=y
+S:CONFIG_TARGET_NOVA=y
```

On the first line, `CONFIG_SPL=y` causes the SPL binary, MLO, to be generated, `CONFIG_ARM=y` causes the contents of `arch/arm/Kconfig` to be included on line three. On line four, `CONFIG_TARGET_NOVA=y` selects your board. Note that lines three and four are prefixed by `+S:` so that they apply to both the SPL and normal binaries.

You should also add a menu option to the ARM architecture `Kconfig` that allows people to select Nova as a target:

```
CONFIG_SPL=y
config TARGET_NOVA
bool "Support Nova!"
```

Board-specific files

Each board has a subdirectory named `board/ [board name]` or `board/ [vendor] / [board name]` which should contain:

- `Kconfig`: Contains configuration options for the board
- `MAINTAINERS`: Contains a record of whether the board is currently maintained and, if so, by whom
- `Makefile`: Used to build the board-specific code
- `README`: Contains any useful information about this port of U-Boot, for example, which hardware variants are covered

In addition, there may be source files for board specific functions.

Your Nova board is based on a BeagleBone which, in turn, is based on a TI AM335x EVM, so, you can start by taking a copy of the am335x board files:

```
$ mkdir board/nova
$ cp -a board/ti/am335x board/nova
```

Next, change the `Kconfig` file to reflect the Nova board:

```
if TARGET_NOVA

    config SYS_CPU
    default "armv7"

    config SYS_BOARD
    default "nova"

    config SYS_SOC
    default "am33xx"

    config SYS_CONFIG_NAME
    default "nova"
endif
```

Setting `SYS_CPU` to `armv7` causes the code in `arch/arm/cpu/armv7` to be compiled and linked. Setting `SYS_SOC` to `am33xx` causes the code in `arch/arm/cpu/armv7/am33xx` to be included, setting `SYS_BOARD` to `nova` brings in `board/nova` and setting `SYS_CONFIG_NAME` to `nova` means that the header file `include/configs/nova.h` is used for further configuration options.

There is one other file in `board/nova` that you need to change, the linker script placed at `board/nova/u-boot.lds`, which has a hard-coded reference to `board/ti/am335x/built-in.o`. Change this to use the copy local to `nova`:

```
diff --git a/board/nova/u-boot.lds b/board/nova/u-boot.lds
index 78f294a..6689b3d 100644
--- a/board/nova/u-boot.lds
+++ b/board/nova/u-boot.lds
@@ -36,7 +36,7 @@ SECTIONS
 *(.image_copy_start)
 *(.vectors)
 CPUDIR/start.o (.text*)
- board/ti/am335x/built-in.o (.text*)
+ board/nova/built-in.o (.text*)
*(.text*)
}
```

Configuration header files

Each board has a header file in `include/configs` which contains the majority of the configuration. The file is named by the `SYS_CONFIG_NAME` identifier in the board's `Kconfig`. The format of this file is described in detail in the `README` file at the top level of the U-Boot source tree.

For the purposes of your Nova board, simply copy `am335x_evm.h` to `nova.h` to `nova.h` and make a small number of changes:

```
diff --git a/include/configs/nova.h b/include/configs/nova.h
index a3d8a25..8ea1410 100644
--- a/include/configs/nova.h
+++ b/include/configs/nova.h
@@ -1,5 +1,5 @@
/*
- * am335x_evm.h
+ * nova.h, based on am335x_evm.h
 *
 * Copyright (C) 2011 Texas Instruments Incorporated -
 http://www.ti.com/
 *
```

```
@@ -13,8 +13,8 @@
 * GNU General Public License for more details.
 */
#ifndef __CONFIG_AM335X_EVM_H
#define __CONFIG_AM335X_EVM_H
#ifndef __CONFIG_NOVA
#define __CONFIG_NOVA
#include <configs/ti_am335x_common.h>
@@ -39,7 +39,7 @@
#define V_SCLK (V_OSCK)
/* Custom script for NOR */
#define CONFIG_SYS_LDSCRIPT "board/ti/am335x/u-boot.lds"
#define CONFIG_SYS_LDSCRIPT "board/nova/u-boot.lds"
/* Always 128 KiB env size */
#define CONFIG_ENV_SIZE (128 << 10)
@@ -50,6 +50,9 @@
#define CONFIG_PARTITION_UUIDS
#define CONFIG_CMD_PART
#undef CONFIG_SYS_PROMPT
#define CONFIG_SYS_PROMPT "nova!> "
+
#endif CONFIG_NAND
#define NANDARGS \
"mtdids=" MTDIDS_DEFAULT "\0" \
```

Building and testing

To build for the Nova board, select the configuration you have just created:

```
$ make CROSS_COMPILE=arm-cortex-a8-linux-gnueabi- nova_defconfig
$ make CROSS_COMPILE=arm-cortex-a8-linux-gnueabi-
```

Copy `MLO` and `u-boot.img` to the FAT partition of the micro-SD card you created earlier and boot the board.

Falcon mode

We are used to the idea that booting a modern embedded processor involves the CPU boot ROM loading an SPL which loads `u-boot.bin` which then loads a Linux kernel. You may be wondering if there is a way to reduce the number of steps, thereby simplifying and speeding up the boot process. The answer is U-Boot "Falcon mode", named after the Peregrine falcon which is claimed to be the fastest of all birds.

The idea is simple: have the SPL load a kernel image directly, missing out `u-boot.bin`. There is no user interaction and there are no scripts. It just loads a kernel from a known location in flash or eMMC into memory, passes it a pre-prepared parameter block and starts it running. The details of configuring Falcon mode are beyond this book. If you would like more information, take a look at `doc/README.falcon`.

Barebox

I will complete this chapter with a look at another bootloader that has the same roots as U-Boot but takes a new approach to bootloaders. It is derived from U-Boot and was actually called U-Boot v2 in the early days. The Barebox developers aimed to combine the best parts of U-Boot and Linux, including a POSIX-like API and mountable filesystems.

The Barebox project website is www.barebox.org and the developer mailing list is barebox@lists.infradead.org.

Getting Barebox

To get Barebox, clone the git repository and check out the version you want to use:

```
$ git clone git://git.pengutronix.de/git/barebox.git  
$ cd barebox  
$ git checkout v2014.12.0
```

The layout of the code is similar to U-Boot:

- `arch`: Contains code specific to each supported architecture, which includes all the major embedded architectures. SoC support is in `arch/ [architecture] / mach- [SoC]`. Support for individual boards is in `arch/ [architecture] /boards`.
- `common`: Contains core functions, including the shell.
- `commands`: Contains the commands that can be called from the shell.
- `Documentation`: Contains the templates for documentation files. To build it, type "make docs". The results are put in `Documentation/html`.
- `drivers`: Contains the code for the device drivers.
- `include`: Contains header files.

Building Barebox

Barebox has used kconfig/kbuild for a long time. There are default configuration files in `arch/ [architecture]/configs`. As an example, assume that you want to build Barebox for the BeagleBoard C4. You need two configurations, one for the SPL, and one for the main binary. Firstly, build MLO:

```
$ make ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabi-
omap3530_beagle_xload_defconfig
$ make ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabi-
```

The result is the secondary program loader, MLO.

Next, build Barebox:

```
$ make ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabi-
omap3530_beagle_defconfig
$ make ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabi-
```

Copy both to an SD card:

```
$ cp MLO /media/boot/
$ cp barebox-flash-image /media/boot/barebox.bin
```

Then, boot up the board and you should see messages like these on the console:

```
barebox 2014.12.0 #1 Wed Dec 31 11:04:39 GMT 2014
```

```
Board: Texas Instruments beagle
nand: Trying ONFI probe in 16 bits mode, aborting !
nand: NAND device: Manufacturer ID: 0x2c, Chip ID: 0xba (Micron ),
256MiB, page
size: 2048, OOB size: 64
omap-hsmmc omap3-hsmmc0: registered as omap3-hsmmc0
mci0: detected SD card version 2.0
mci0: registered disk0
malloc space: 0x87bff400 -> 0x87ffff3ff (size 4 MiB)
booting from MMC
```

```
barebox 2014.12.0 #2 Wed Dec 31 11:08:59 GMT 2014
```

```
Board: Texas Instruments beagle
netconsole: registered as netconsole-1
```

```
i2c-omap i2c-omap30: bus 0 rev3.3 at 100 kHz
ehci ehci0: USB EHCI 1.00
nand: Trying ONFI probe in 16 bits mode, aborting !
nand: NAND device: Manufacturer ID: 0x2c, Chip ID: 0xba (Micron NAND
256MiB 1,8V
16-bit), 256MiB, page size: 2048, OOB size: 64
omap-hsmmc omap3-hsmmc0: registered as omap3-hsmmc0
mci0: detected SD card version 2.0
mci0: registered disk0
malloc space: 0x85e00000 -> 0x87dfffff (size 32 MiB)
environment load /boot/barebox.env: No such file or directory
Maybe you have to create the partition.
no valid environment found on /boot/barebox.env. Using default
environment
running /env/bin/init...
```

Hit any key to stop autoboot: 0

Barebox is continuing to evolve. At the time of writing, it lacks the breadth of hardware support that U-Boot has, but it is worth considering for new projects.

Summary

Every system needs a bootloader to bring the hardware to life and to load a kernel. U-Boot has found favor with many developers because it supports a useful range of hardware and it is fairly easy to port to a new device. Over the last few years, the complexity and ever increasing variety of embedded hardware has led to the introduction of the device tree as a way of describing hardware. The device tree is simply a textual representation of a system that is compiled into a **device tree binary (dtb)** and which is passed to the kernel when it loads. It is up to the kernel to interpret the device tree and to load and initialize drivers for the devices it finds there.

In use, U-Boot is very flexible, allowing images to be loaded from mass storage, flash memory, or a network, and booted. Likewise, Barebox can achieve the same but with a smaller base of hardware support. Despite its cleaner design and POSIX-inspired internal APIs, at the time of writing it does not seem to have been accepted beyond its own small but dedicated community.

Having covered some of the intricacies of booting Linux, in the next chapter you will see the next stage of the process as the third element of your embedded project, the kernel, comes into play.

4

Porting and Configuring the Kernel

The kernel is the third element of embedded Linux. It is the component that is responsible for managing resources and interfacing with hardware and so affects almost every aspect of your final software build. It is usually tailored to your particular hardware configuration although, as we saw in *Chapter 3, All About Bootloaders*, device trees allow you to create a generic kernel that is tailored to particular hardware by the contents of the device tree.

In this chapter, we will look at how to get a kernel for a board and how to configure and compile it. We will look again at bootstrap, this time focusing on the part the kernel plays. We will also look at device drivers and how they pick up information from the device tree.

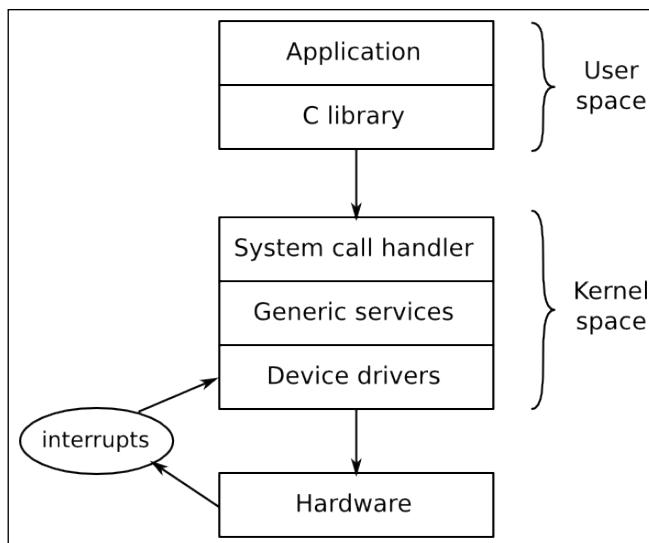
What does the kernel do?

Linux began in 1991 when Linus Torvalds started writing an operating system for Intel 386 and 486-based personal computers. He was inspired by the Minix operating system written by Andrew S. Tanenbaum four years earlier. Linux differed in many ways from Minix, the main differences being that it was a 32-bit virtual memory kernel and the code was open source, later released under the GPL 2 license.

He announced it on the 25th August 1991 on the *comp.os.minix* newsgroup in a famous post that begins as *Hello everybody out there using minix - I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since april, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things).*

To be strictly accurate, Linus did not write an operating system, he wrote a kernel instead, which is one component of an operating system. To create a working system, he used components from the GNU project, especially the toolchain, C library, and basic command-line tools. That distinction remains today, and gives Linux a lot of flexibility in the way it is used. It can be combined with a GNU user space to create a full Linux distribution that runs on desktops and servers, which is sometimes called GNU/Linux; it can be combined with an Android user space to create the well-known mobile operating system or it can be combined with a small Busybox-based user space to create a compact embedded system. Contrast this with the BSD operating systems, FreeBSD, OpenBSD, and NetBSD, in which the kernel, the toolchain, and the user space are combined into a single code base.

The kernel has three main jobs: to manage resources, to interface with hardware, and to provide an API that offers a useful level of abstraction to user space programs, as summarized in the following diagram:



Applications running in user space run at a low CPU privilege level. They can do very little other than make library calls. The primary interface between the user space and the kernel space is the C library, which translates user level functions such as those defined by POSIX into kernel system calls. The system call interface uses an architecture-specific method such as a trap or a software interrupt to switch the CPU from the low privilege user mode to the high privilege kernel mode, which allows access to all memory addresses and CPU registers.

The system call handler dispatches the call to the appropriate kernel subsystem: scheduling calls to the scheduler, the filesystem, calls to the filesystem code, and so on. Some of those calls require input from the underlying hardware and will be passed down to a device driver. In some cases, the hardware itself invokes a kernel function by raising an interrupt. Interrupts can only be handled in a device driver, never by a user space application.

In other words, all the useful things that your application does, it does them through the kernel. The kernel, then, is one of the most important elements in the system.

Choosing a kernel

The next step is to choose the kernel for your project, balancing the desire to always use the latest version of software against the need for vendor-specific additions.

Kernel development cycle

Linux has been developed at a fast pace, with a new version being released every 8 to 12 weeks. The way that the version numbers are constructed has changed a bit in recent years. Before July 2011, there was a three number version scheme with version numbers that looked like 2.6.39. The middle number indicated whether it was a developer or stable release, odd numbers (2.1.x, 2.3.x, 2.5.x) were for developers and even numbers were for end users. From version 2.6 onwards, the idea of a long-lived development branch (the odd numbers) was dropped as it slowed down the rate at which new features were made available to users. The change in numbering from 2.6.39 to 3.0 in July 2011 was purely because Linus felt that the numbers were becoming too large: there was no huge leap in the features or architecture of Linux between those two versions. He also took the opportunity to drop the middle number. Since then, in April 2015, he bumped the major from 3 to 4, again purely for neatness, not because of any large architectural shift.

Linus manages the development kernel tree. You can follow him by cloning his git tree like so:

```
$ git clone \
git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
```

This will check out into subdirectory `linux`. You can keep up to date by running the command `git pull` in that directory from time to time.

Currently, a full cycle of kernel development begins with a merge window of two weeks, during which Linus will accept patches for new features. At the end of the merge window, a stabilization phase begins, during which Linus will produce release candidates with version numbers ending in -rc1, -rc2, and so on, usually up to -rc7 or -rc8. During this time, people test the candidates and submit bug reports and fixes. When all significant bugs have been fixed, the kernel is released.

The code incorporated during the merge window has to be fairly mature already. Usually, it is pulled from the repositories of the many subsystem and architecture maintainers of the kernel. By keeping to a short development cycle, features can be merged when they are ready. If a feature is deemed not sufficiently stable or well developed by the kernel maintainers, it can simply be delayed until the next release.

Keeping a track of what has changed from release to release is not easy. You can read the commit log in Linus' git repository but, with roughly 10,000 or more entries per release, it is not easy to get an overview. Thankfully, there is the *Linux Kernel Newbies* website, <http://kernelnewbies.org> where you will find a succinct overview of each version, at <http://kernelnewbies.org/LinuxVersions>.

Stable and long term support releases

The rapid rate of change of Linux is a good thing in that it brings new features into the mainline code base, but it does not fit very well with the longer life cycle of embedded projects. Kernel developers address this in two ways. Firstly, it is acknowledged that a release may contain bugs that need to be fixed before the next kernel release comes around. That is the role of the stable Linux kernel, maintained by Greg Kroah-Hartman. After release, the kernel moves from being **mainline** (maintained by Linus) to **stable** (maintained by Greg). Bug fix releases of the stable kernel are marked by a third number, 3.18.1, 3.18.2, and so on. Before version 3, there were four release numbers, 2.6.29.1, 2.6.39.2, and so on.

You can get the stable tree by using the following command:

```
$ git clone \
git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-
stable.git
```

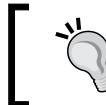
You can use `git checkout` to get a particular version, for example version 4.1.10:

```
$ cd linux-stable
$ git checkout v4.1.10
```

Usually, the stable kernel is maintained only until the next mainline release, 8 to 12 weeks later, so you will see that there is just one or sometimes two stable kernels at `kernel.org`. To cater for those users who would like updates for a longer period of time and be assured that any bugs will be found and fixed, some kernels are labeled **long term** and maintained for two or more years. There is at least one long term kernel each year. Looking at `kernel.org` at the time of writing, there are a total of eight long term kernels: 4.1, 3.18, 3.14, 3.12, 3.10, 3.4, 3.2, and 2.6.32. The latter has been maintained for five years and is at version 2.6.32.68. If you are building a product that you will have to maintain for this length of time the latest long term kernel might well be a good choice.

Vendor support

In an ideal world, you would be able to download a kernel from `kernel.org` and configure it for any device that claims to support Linux. However, that is not always possible: in fact mainline Linux has solid support for only a small subset of the many devices that can run Linux. You may find support for your board or SoC from independent open source projects, Linaro or the Yocto Project, for example, or from companies providing third party support for embedded Linux, but in many cases you will be obliged to look to the vendor of your SoC or board for a working kernel. As we also know, some are better than others.



My only advice at this point is to choose vendors who give good support or who, even better, take the trouble to get their kernel changes into the mainline.

Licensing

The Linux source code is licensed under GPL v2, which means that you must make the source code of your kernel available in one of the ways specified in the license.

The actual text of the license for the kernel is in the file `COPYING`. It begins with an addendum written by Linus that states that code calling the kernel from user space via the system call interface is not considered a derivative work of the kernel and so is not covered by the license. Hence, there is no problem with proprietary applications running on top of Linux.

However, there is one area of Linux licensing that causes endless confusion and debate: kernel modules. A kernel module is simply a piece of code that is dynamically linked with the kernel at runtime, thereby extending the functionality of the kernel. The GPL makes no distinction between static and dynamic linking, so it would appear that the source for kernel modules is covered by the GPL. But, in the early days of Linux, there were debates about exceptions to this rule, for example, in connection with the Andrew filesystem. This code predates Linux and therefore (it was argued) is not a derivative work, and so the license does not apply. Similar discussions took place over the years with respect to other pieces of code, with the result that it is now accepted practice that the GPL does not necessarily apply to kernel modules. This is codified by the kernel `MODULE_LICENSE` macro, which may take the value `Proprietary` to indicate that it is not released under the GPL. If you plan to use the same arguments yourself, you may want to read through an oft-quoted email thread titled *Linux GPL and binary module exception clause?* (http://yarchive.net/comp/linux/gpl_modules.html).

The GPL should be considered a good thing because it guarantees that when you and I are working on embedded projects, we can always get the source code for the kernel. Without it, embedded Linux would be much harder to use and more fragmented.

Building the kernel

Having decided which kernel to base your build on, the next step is to build it.

Getting the source

Let's assume that you have a board that is supported in mainline. You can get the source code through git or by downloading a tarball. Using git is better because you can see the commit history, you can easily see any changes you may make and you can switch between branches and versions. In this example, we are cloning the stable tree and checking out the version tag 4.1.10:

```
$ git clone  
git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-  
stable.git linux  
$ cd linux  
$ git checkout v4.1.10
```

Alternatively, you could download the tarball from <https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.1.10.tar.xz>.

There is a lot of code here. There are over 38,000 files in the 4.1 kernel containing C source code, header files, and assembly code, amounting to a total of over 12.5 million lines of code (as measured by the cloc utility). Nevertheless, it is worth knowing the basic layout of the code and to know, approximately, where to look for a particular component. The main directories of interest are:

- `arch`: This contains architecture-specific files. There is one subdirectory per architecture.
- `Documentation`: This contains kernel documentation. Always look here first if you want to find more information about an aspect of Linux.
- `drivers`: This contains device drivers, thousands of them. There is a subdirectory for each type of driver.
- `fs`: This contains filesystem code.
- `include`: This contains kernel header files, including those required when building the toolchain.
- `init`: This contains the kernel start-up code.
- `kernel`: This contains core functions, including scheduling, locking, timers, power management, and debug/trace code.
- `mm`: This contains memory management.
- `net`: This contains network protocols.
- `scripts`: This contains many useful scripts including the device tree compiler, dtc, which I described in *Chapter 3, All About Bootloaders*.
- `tools`: This contains many useful tools, including the Linux performance counters tool, perf, which I will describe in *Chapter 13, Profiling and Tracing*.

Over a period of time, you will become familiar with this structure, and realize that, if you are looking for the code for the serial port of a particular SoC, you will find it in `drivers/tty/serial` and not in `arch/$ARCH/mach-foo` because it is a device driver and not something central to the running of Linux on that SoC.

Understanding kernel configuration

One of the strengths of Linux is the degree to which you can configure the kernel to suit different jobs, from a small dedicated device such as a smart thermostat to a complex mobile handset. In current versions there are many thousands of configuration options. Getting the configuration right is a task in itself but, before that, I want to show you how it works so that you can better understand what is going on.

The configuration mechanism is called `Kconfig`, and the build system that it integrates with is called `Kbuild`. Both are documented in `Documentation/kbuild/`. `Kconfig/Kbuild` is used in a number of other projects as well as the kernel, including `crosstool-NG`, `U-Boot`, `Barebox`, and `BusyBox`.

The configuration options are declared in a hierarchy of files named `Kconfig` using a syntax described in `Documentation/kbuild/kconfig-language.txt`. In Linux, the top level `Kconfig` looks like this:

```
mainmenu "Linux/$ARCH $KERNELVERSION Kernel Configuration"
config SRCARCH
    string
    option env="SRCARCH"
    source "arch/$SRCARCH/Kconfig"
```

The last line includes the architecture-dependent configuration file which sources other `Kconfig` files depending on which options are enabled. Having the architecture play such a role has two implications: firstly, that you must specify an architecture when configuring Linux by setting `ARCH=[architecture]`, otherwise it will default to the local machine architecture, and second that the layout of the top level menu is different for each architecture.

The value you put into `ARCH` is one of the subdirectories you find in directory `arch`, with the oddity that `ARCH=i386` and `ARCH=x86_64` both have the source `arch/x86/Kconfig`.

The `Kconfig` files consist largely of menus, delineated by `menu`, `menu title`, and `endmenu` keywords, and menu items marked by `config`. Here is an example, taken from `drivers/char/Kconfig`:

```
menu "Character devices"
[...]
config DEVMEM
    bool "/dev/mem virtual device support"
    default y
    help
        Say Y here if you want to support the /dev/mem device.
        The /dev/mem device is used to access areas of physical
        memory.
        When in doubt, say "Y".
```

The parameter following `config` names a variable that, in this case, is `DEVMEM`. Since this option is a Boolean, it can only have two values: if it is enabled it is assigned to `y`, if not the variable is not defined at all. The name of the menu item that is displayed on the screen is the string following the `bool` keyword.

This configuration item, along with all the others, is stored in a file named `.config` (note that the leading dot '.' means that it is a hidden file that will not be shown by the `ls` command unless you type `ls -a` to show all files). The variable names stored in `.config` are prefixed with `CONFIG_`, so if `DEVMEM` is enabled, the line reads:

```
CONFIG_DEVMEM=y
```

There are several other data types in addition to `bool`. Here is the list:

- `bool`: This is either `y` or not defined.
- `tristate`: This is used where a feature can be built as a kernel module or built into the main kernel image. The values are `m` for a module, `y` to be built in, and not defined if the feature is not enabled.
- `int`: This is an integer value written using decimal notation.
- `hex`: This is an unsigned integer value written using hexadecimal notation.
- `string`: This is a string value.

There may be dependencies between items, expressed by the `depends on` phrase, as shown here:

```
config MTD_CMDLINE_PARTS
    tristate "Command line partition table parsing"
    depends on MTD
```

If `CONFIG_MTD` has not been enabled elsewhere, this menu option is not shown and so cannot be selected.

There are also reverse dependencies: the `select` keyword enables other options if this one is enabled. The `Kconfig` file in `arch/$ARCH` has a large number of `select` statements that enable features specific to the architecture, as can be seen here for arm:

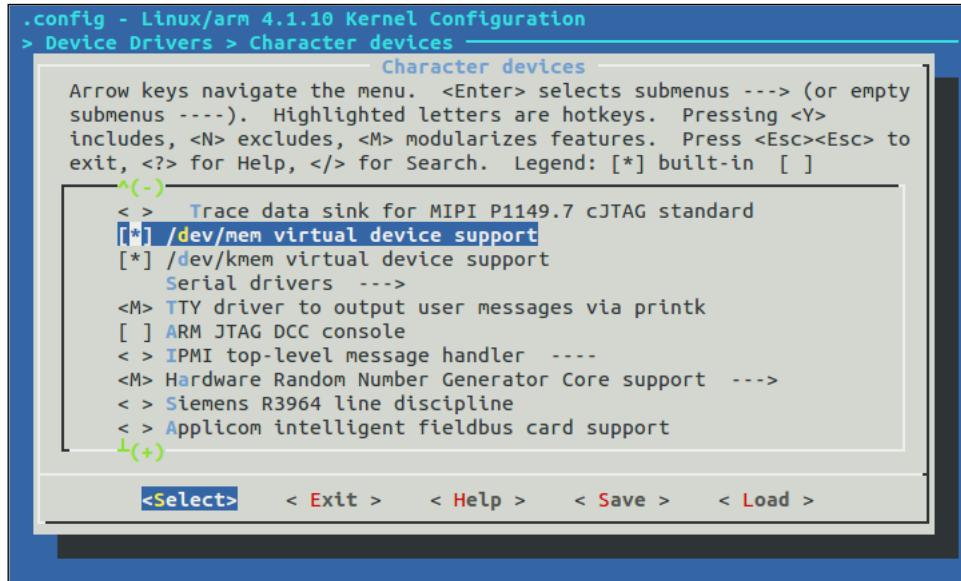
```
config ARM
    bool
    default y
    select ARCH_HAS_ATOMIC64_DEC_IF_POSITIVE
    select ARCH_HAS_ELF_RANDOMIZE
    [...]
```

There are several configuration utilities that can read the `Kconfig` files and produce a `.config` file. Some of them display the menus on screen and allow you to make choices interactively. `Menuconfig` is probably the one most people are familiar with, but there is also `xconfig` and `gconfig`.

You launch each one via `make`, remembering that, in the case of the kernel, you have to supply an architecture, as illustrated here:

```
$ make ARCH=arm menuconfig
```

Here, you can see `menuconfig` with the `DEVMEM` config option highlighted in the previous paragraph:



Kernel configuration using `menuconfig`

The star (*) to the left of an item means that it is selected (= "y") or, if it is an M, that it has been selected to be built as a kernel module.



You often see instructions like `enable CONFIG_BLK_DEV_INITRD`, but with so many menus to browse through, it can take a while to find the place where that configuration is set. All configuration editors have a search function. You can access it in `menuconfig` by pressing the forward slash key, /. In `xconfig`, it is in the edit menu but, in this case make sure you miss off the `CONFIG_` part of the variable you are searching for.

With so many things to configure, it is unreasonable to start with a clean sheet each time you want to build a kernel so there are a set of known working configuration files in `arch/$ARCH/configs`, each containing suitable configuration values for a single SoC or a group of SoCs. You can select one with `make [configuration file name]`. For example, to configure Linux to run on a wide range of SoCs using the armv7-a architecture, which includes the BeagleBone Black AM335x, you would type:

```
$ make ARCH=arm multi_v7_defconfig
```

This is a generic kernel that runs on various different boards. For a more specialized application, for example when using a vendor-supplied kernel, the default configuration file is part of the board support package; you will need to find out which one to use before you can build the kernel.

There is another useful configuration target named `oldconfig`. This takes an existing `.config` file and asks you to supply configuration values for any options that don't have them. You would use it when moving a configuration to a newer kernel version: copy `.config` from the old kernel to the new source directory and run `make ARCH=arm oldconfig` to bring it up to date. It can also be used to validate a `.config` file that you have edited manually (ignoring the text `Automatically generated file; DO NOT EDIT` that occurs at the top: sometimes it is OK to ignore warnings).

If you do make changes to the configuration, the modified `.config` file becomes part of your device and needs to be placed under source code control.

When you start the kernel build, a header file, `include/generated/autoconf.h`, is generated which contains a `#define` for each configuration value so that it can be included in the kernel source, exactly as with U-Boot.

Using LOCALVERSION to identify your kernel

You can discover the kernel version that you have built using the `make kernelversion` target:

```
$ make kernelversion  
4.1.10
```

This is reported at runtime through the `uname` command and is also used in naming the directory where kernel modules are stored.

If you change the configuration from the default it is advisable to append your own version information, which you can configure by setting `CONFIG_LOCALVERSION`, which you will find in the **General setup configuration** menu. It is also possible (but discouraged) to do the same by editing the top level makefile and appending it to the line that begins with `EXTRAVERSION`. As an example, if I wanted to mark the kernel I am building with an identifier `melp` and version 1.0, I would define the local version in the `.config` file like this:

```
CONFIG_LOCALVERSION="-melp-v1.0"
```

Running `make kernelversion` produces the same output as before but now, if I run `make kernelrelease`, I see:

```
$ make kernelrelease  
4.1.10-melp-v1.0
```

It is also printed at the beginning of the kernel log:

```
Starting kernel ...  
[    0.000000] Booting Linux on physical CPU 0x0  
[    0.000000] Linux version 4.1.10-melp-v1.0 (chris@builder) (gcc  
version 4.9.1 (crosstool-NG 1.20.0) ) #3 SMP Thu Oct 15 21:29:35 BST 2015
```

I can now identify and track my custom kernel.

Kernel modules

I have mentioned kernel modules several times already. Desktop Linux distributions use them extensively so that the correct device and kernel functions can be loaded at runtime depending on the hardware detected and features required. Without them, every single driver and feature would have to be statically linked in to the kernel, making it unfeasibly large.

On the other hand, with embedded devices, the hardware and kernel configuration is usually known at the time the kernel is built so modules are not so useful. In fact, they cause a problem because they create a version dependency between the kernel and the root filesystem which can cause boot failures if one is updated but not the other. Consequently, it is quite common for embedded kernels to be built without any modules at all. Here are a few cases where kernel modules are a good idea:

- When you have proprietary modules, for the licensing reasons given in the preceding section.
- To reduce boot time by deferring the loading of non-essential drivers.

- When there are a number of drivers that could be loaded and it would take up too much memory to compile them statically. For example, you have a USB interface to support a range of devices. This is essentially the same argument as is used in desktop distributions.

Compiling

The kernel build system, `kbuild`, is a set of `make` scripts that take the configuration information from the `.config` file, work out the dependencies and compile everything that is necessary to produce a kernel image containing all the statically linked components, possibly a device tree binary and possibly one or more kernel modules. The dependencies are expressed in the makefiles that are in each directory with buildable components. For instance, these two lines are taken from `drivers/char/Makefile`:

```
obj-y                  += mem.o random.o
obj-$(CONFIG_TTY_PRINTK) += ttyprintk.o
```

The `obj-y` rule unconditionally compiles a file to produce the target, so `mem.c` and `random.c` are always part of the kernel. In the second line, `ttyprintk.c` is dependent on a configuration parameter. If `CONFIG_TTY_PRINTK` is `y` it is compiled as a built in, if it is `m` it is built as a module and, if the parameter is undefined, it is not compiled at all.

For most targets, just typing `make` (with the appropriate `ARCH` and `CROSS_COMPILE`) will do the job, but it is instructive to take it one step at a time.

Compiling the kernel image

To build a kernel image, you need to know what your bootloader expects. This is a rough guide:

- **U-Boot:** Traditionally U-Boot has required a `uImage`, but newer versions can load a `zImage` file using the `bootz` command
- **x86 targets:** It requires a `bzImage` file
- **Most other bootloaders:** It requires a `zImage` file

Here is an example of building a `zImage` file:

```
$ make -j 4 ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf-  
zImage
```



The `-j 4` option tells `make` how many jobs to run in parallel, which reduces the time taken to build. A rough guide is to run as many jobs as you have CPU cores.



It is the same when building `bzImage` and `uImage` targets.

There is a small issue with building a `uImage` file for ARM with multi-platform support, which is the norm for the current generation of ARM SoC kernels. Multi-platform support for ARM was introduced in Linux 3.7. It allows a single kernel binary to run on multiple platforms and is a step on the road toward having a small number of kernels for all ARM devices. The kernel selects the correct platform by reading the machine number or the device tree passed to it by the bootloader. The problem occurs because the location of physical memory might be different for each platform, and so the relocation address for the kernel (usually 0x8000 bytes from the start of physical RAM) might also be different. The relocation address is coded into the `uImage` header by the `mkimage` command when the kernel is built, but it will fail if there is more than one relocation address to choose from. To put it another way, the `uImage` format is not compatible with multi-platform images. You can still create a `uImage` binary from a multi-platform build so long as you give the `LOADADDR` of the particular SoC you are hoping to boot this kernel on. You can find the load address by looking in `mach-[your SoC]/Makefile.boot` and noting the value of `zreladdr-y`.

In the case of a BeagleBone Black, the full command would look like this:

```
$ make -j 4 ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf-  
LOADADDR=0x80008000 uImage
```

A kernel build generates two files in the top level directory: `vmlinu`x and `System.map`. The first, `vmlinu`x, is the kernel as an ELF binary. If you have compiled your kernel with debug enabled (`CONFIG_DEBUG_INFO=y`), it will contain debug symbols which can be used with debuggers like kgdb. You can also use other ELF binary tools such as `size`:

```
$ arm-cortex_a8-linux-gnueabihf-size vmlinu
      text      data      bss      dec      hex   filename
8812564    790692   8423536  18026792  1131128   vmlinu
```

`System.map` contains the symbol table in human readable form.

Most bootloaders cannot handle ELF code directly. There is a further stage of processing which takes `vmlinu`x and places those binaries in `arch/$ARCH/boot` that are suitable for the various bootloaders:

- `Image`: `vmlinu`x converted to raw binary.
- `zImage`: For the PowerPC architecture, this is just a compressed version of `Image`, which implies that the bootloader must do the decompression. For all other architectures, the compressed `Image` is piggybacked onto a stub of code that decompresses and relocates it.
- `uImage`: `zImage` plus a 64-byte U-Boot header.

While the build is running, you will see a summary of the commands being executed:

```
$ make -j 4 ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf-
zImage
CC      init/main.o
CHK    include/generated/compile.h
CC      init/version.o
CC      init/do_mounts.o
CC      init/do_mounts_rd.o
CC      init/do_mounts_initrd.o
LD      init/mounts.o
[...]
```

Sometimes, when the kernel build fails, it is useful to see the actual commands being executed. To do that, add `v=1` to the command line:

```
$ make ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf- V=1
zImage
[...]
arm-cortex_a8-linux-gnueabihf-gcc -Wp,-
MD,init/.do_mounts_initrd.o.d -nostdinc -isystem /home/chris/x-
tools/arm-cortex_a8-linux-gnueabihf/lib/gcc/arm-cortex_a8-linux-
gnueabihf/4.9.1/include -I./arch/arm/include -
Iarch/arm/include/generated/uapi -Iarch/arm/include/generated -
Iinclude -I./arch/arm/include/uapi -
Iarch/arm/include/generated/uapi -I./include/uapi -
Iinclude/generated/uapi -include ./include/linux/kconfig.h -
D_KERNEL__ -mlittle-endian -Wall -Wundef -Wstrict-prototypes -
Wno-trigraphs -fno-strict-aliasing -fno-common -Werror-implicit-
function-declaration -Wno-format-security -std=gnu89 -fno-dwarf2-
cfi-asm -mabi=aapcs-linux -mno-thumb-interwork -mfpu=vfp -funwind-
tables -marm -D_LINUX_ARM_ARCH_=7 -march=armv7-a -msoft-float -
Uarm -fno-delete-null-pointer-checks -O2 --param=allow-store-data-
races=0 -Wframe-larger-than=1024 -fno-stack-protector -Wno-unused-
but-set-variable -fomit-frame-pointer -fno-var-tracking-
assignments -Wdeclaration-after-statement -Wno-pointer-sign -fno-
strict-overflow -fconserve-stack -Werror=implicit-int -
Werror=strict-prototypes -Werror=date-time -DCC_HAVE_ASM_GOTO -
D"KBUILD_STR(s)="#s" -
D"KBUILD_BASENAME=KBUILD_STR(do_mounts_initrd)" -
D"KBUILD_MODNAME=KBUILD_STR(mounts)" -c -o init/do_mounts_initrd.o
init/do_mounts_initrd.c
[...]
```

Compiling device trees

The next step is to build the device tree, or trees if you have a multi-platform build. The `dtbs` target builds device trees according to the rules in `arch/$ARCH/boot/dts/Makefile` using the device tree source files in that directory:

```
$ make ARCH=arm dtbs
...
DTC    arch/arm/boot/dts/omap2420-h4.dtb
DTC    arch/arm/boot/dts/omap2420-n800.dtb
DTC    arch/arm/boot/dts/omap2420-n810.dtb
DTC    arch/arm/boot/dts/omap2420-n810-wimax.dtb
DTC    arch/arm/boot/dts/omap2430-sdp.dtb
...
```

The `.dtb` files are generated in the same directory as the sources.

Compiling modules

If you have configured some features to be built as modules, you can build them separately using the `modules` target:

```
$ make -j 4 ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf-  
modules
```

The compiled modules have a `.ko` suffix and are generated in the same directory as the source code, meaning that they are scattered all around the kernel source tree. Finding them is a little tricky but you can use the `modules_install` make target to install them in the right place. The default location is `/lib/modules` in your development system, which is almost certainly not what you want. To install them into the staging area of your root filesystem (we will talk about root filesystems in the next chapter), provide the path using `INSTALL_MOD_PATH`:

```
$ make -j4 ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf-  
INSTALL_MOD_PATH=$HOME/rootfs modules_install
```

Kernel modules are put into the directory `/lib/modules/[kernel version]`, relative to the root of the filesystem.

Cleaning kernel sources

There are three make targets for cleaning the kernel source tree:

- `clean`: removes object files and most intermediates.
- `mrproper`: removes all intermediate files, including the `.config` file. Use this target to return the source tree to the state it was in immediately after cloning or extracting the source code. If you are curious about the name, Mr Proper is a cleaning product common in some parts of the world. The meaning of `make mrproper` is to give the kernel sources a really good scrub.
- `districlean`: This is the same as `mrproper` but also deletes editor backup files, patch leftover files, and other artifacts of software development.

Booting your kernel

Booting is highly device-dependent, but here is an example using U-Boot on a BeagleBone Black and QEMU::

BeagleBone Black

The following U-Boot commands show how to boot Linux on a BeagleBone Black:

```
U-Boot# fatload mmc 0:1 0x80200000 zImage
reading zImage
4606360 bytes read in 254 ms (17.3 MiB/s)
U-Boot# fatload mmc 0:1 0x80f00000 am335x-boneblack.dtb
reading am335x-boneblack.dtb
29478 bytes read in 9 ms (3.1 MiB/s)
U-Boot# setenv bootargs console=ttyO0,115200
U-Boot# bootz 0x80200000 - 0x80f00000
Kernel image @ 0x80200000 [ 0x000000 - 0x464998 ]
## Flattened Device Tree blob at 80f00000
    Booting using the fdt blob at 0x80f00000
    Loading Device Tree to 8fff5000, end 8ffff325 ... OK
Starting kernel ...
[    0.000000] Booting Linux on physical CPU 0x0
...

```

Note that we set the kernel command line to `console=ttyO0,115200`. That tells Linux which device to use for console output which, in this case, is the first UART on the board, device `ttyO0`, at a speed of 115,200 bits per second. Without this, we would not see any messages after `Starting kernel ...` and therefore would not know if it was working or not.

QEMU

Assuming that you have already installed `qemu-system-arm`, you can launch it with the `multi_v7` kernel and the `.dtb` file for the ARM Versatile Express, as follows:

```
$ QEMU_AUDIO_DRV=none \
qemu-system-arm -m 256M -nographic -M vexpress-a9 -kernel zImage -
dtb vexpress-v2p-ca9.dtb -append "console=ttyAMA0"
```

Note that setting `QEMU_AUDIO_DRV` to `none` is just to suppress error messages from QEMU about missing configurations for the audio drivers, which we do not use.

To exit from QEMU, type `Ctrl-A` then `x` (two separate keystrokes).

Kernel panic

While things started off well, they ended badly:

```
[    1.886379] Kernel panic - not syncing: VFS: Unable to mount  
root fs on unknown-block(0,0)  
[    1.895105] ---[ end Kernel panic - not syncing: VFS: Unable to  
mount root fs on unknown-block(0, 0)
```

This is a good example of a kernel panic. A panic occurs when the kernel encounters an unrecoverable error. By default, it will print out a message to the console and then halt. You can set the `panic` command line parameter to allow a few seconds before it reboots following a panic.

In this case, the unrecoverable error is because there is no root filesystem, illustrating that a kernel is useless without a user space to control it. You can supply a user space by providing a root filesystem either as a ramdisk or on a mountable mass storage device. We will talk about how to create a root filesystem in the next chapter but, to get things up and running, assume that we have a ramdisk in the file `uRamdisk` and you can then boot to a shell prompt by entering these commands into U-Boot:

```
fatload mmc 0:1 0x80200000 zImage  
fatload mmc 0:1 0x80f00000 am335x-boneblack.dtb  
fatload mmc 0:1 0x81000000 uRamdisk  
setenv bootargs console=tty00,115200 rdinit=/bin/sh  
bootz 0x80200000 0x81000000 0x80f00000
```

Here, I have added `rdinit=/bin/sh` to the command line so that the kernel will run a shell and give us a shell prompt. Now, the output on the console looks like this:

```
...  
[    1.930923] sr_init: No PMIC hook to init smartreflex  
[    1.936424] sr_init: platform driver register failed for SR  
[    1.964858] Freeing unused kernel memory: 408K (c0824000 -  
c088a000)  
/ # uname -a  
Linux (none) 3.18.3 #1 SMP Wed Jan 21 08:34:58 GMT 2015 armv7l  
GNU/Linux  
/ #
```

At last, we have a prompt and can interact with our device.

Early user space

In order to transition from kernel initialization to user space, the kernel has to mount a root filesystem and execute a program in that root filesystem. This can be via a ramdisk, as shown in the previous section, or by mounting a real filesystem on a block device. The code for all of this is in `init/main.c`, starting with the function `rest_init()` which creates the first thread with PID 1 and runs the code in `kernel_init()`. If there is a ramdisk, it will try to execute the program `/init`, which will take on the task of setting up the user space.

If it fails to find and run `/init`, it tries to mount a filesystem by calling the function `prepare_namespace()` in `init/do_mounts.c`. This requires a `root=` command line to give the name of the block device to use for mounting, usually in the form:

- `root=/dev/<disk name><partition number>`
- `root=/dev/<disk name>p<partition number>`

For example, for the first partition on an SD card, that would be `root=/dev/mmcblk0p1`. If the mount succeeds, it will try to execute `/sbin/init`, followed by `/etc/init`, `/bin/init`, and then `/bin/sh`, stopping at the first one that works.

The `init` program can be overridden on the command line. For a ramdisk, use `rdinit=`, (I used `rdinit=/bin/sh` earlier to execute a shell) and, for a filesystem, use `init=`.

Kernel messages

Kernel developers are fond of printing out useful information through liberal use of `printk()` and similar functions. The messages are categorized according to importance, 0 being the highest:

Level	Value	Meaning
KERN_EMERG	0	The system is unusable
KERN_ALERT	1	Action must be taken immediately
KERN_CRIT	2	Critical conditions
KERN_ERR	3	Error conditions
KERN_WARNING	4	Warning conditions
KERN_NOTICE	5	Normal but significant conditions
KERN_INFO	6	Informational
KERN_DEBUG	7	Debug-level messages

They are first written to a buffer, `__log_buf`, the size of which is two to the power of `CONFIG_LOG_BUF_SHIFT`. For example, if it is 16, then `__log_buf` is 64 KiB. You can dump the entire buffer using the command `dmesg`.

If the level of a message is less than the console log level, it is displayed on the console as well as being placed in `__log_buf`. The default console log level is 7, meaning that messages of level 6 and lower are displayed, filtering out `KERN_DEBUG` which is level 7. You can change the console log level in several ways, including by using the kernel parameter `loglevel=<level>` or the command `dmesg -n <level>`.

Kernel command line

The kernel command line is a string that is passed to the kernel by the bootloader, via the `bootargs` variable in the case of U-Boot; it can also be defined in the device tree, or set as part of the kernel configuration in `CONFIG_CMDLINE`.

We have seen some examples of the kernel command line already but there are many more. There is a complete list in `Documentation/kernel-parameters.txt`. Here is a smaller list of the most useful ones:

Name	Description
<code>debug</code>	Sets the console log level to the highest level, eight, to ensure that you see all the kernel messages on the console.
<code>init=</code>	The <code>init</code> program to run from a mounted root filesystem, which defaults to <code>/sbin/init</code> .
<code>lpj=</code>	Sets the <code>loops_per_jiffy</code> to a given constant, see the following paragraph.
<code>panic=</code>	Behavior when the kernel panics: if it is greater than zero, it gives the number of seconds before rebooting; if it is zero, it waits forever (this is the default); or if it is less than zero, it reboots without any delay.
<code>quiet</code>	Sets the console log level to one, suppressing all but emergency messages. Since most devices have a serial console, it takes time to output all those strings. Consequently, reducing the number of messages using this option reduces boot time.
<code>rdinit=</code>	The <code>init</code> program to run from a ramdisk, it defaults to <code>/init</code> .
<code>ro</code>	Mounts the root device as read-only. Has no effect on a ramdisk which is always read/write.
<code>root=</code>	Device to mount the root filesystem.
<code>rootdelay=</code>	The number of seconds to wait before trying to mount the root device, defaults to zero. Useful if the device takes time to probe the hardware, but also see <code>rootwait</code> .

Name	Description
<code>rootfstype=</code>	The filesystem type for the root device. In many cases, it is auto-detected during mount, but it is required for <code>jffs2</code> filesystems.
<code>rootwait</code>	Waits indefinitely for the root device to be detected. Usually necessary with <code>mmc</code> devices.
<code>rw</code>	Mounts the root device as read-write (default).

The `lpj` parameter is often mentioned in connection with reducing the kernel boot time. During initialization, the kernel loops for approximately 250 ms to calibrate a delay loop. The value is stored in the variable `loops_per_jiffy`, and reported like this:

```
Calibrating delay loop... 996.14 BogoMIPS (lpj=4980736)
```

If the kernel always runs on the same hardware it will always calculate the same value. You can shave 250 ms off the boot time by adding `lpj=4980736` to the command line.

Porting Linux to a new board

The scope of the task depends on how similar your board is to an existing development board. In *Chapter 3, All About Bootloaders* we ported U-Boot to a new board, named Nova, which is based on the BeagleBone Black (when I say based, it actually is one) so, in this case, there are very few changes to the kernel code to be made. If you are porting to completely new and innovative hardware, there will be more to do. I am only going to consider the simple case.

The organization of architecture-specific code in `arch/$ARCH` differs from one system to another. The x86 architecture is pretty clean because hardware details are detected at runtime. The PowerPC architecture puts SoC and board-specific files in subdirectory `platforms`. The ARM architecture has the most board and SoC-specific files of all because there are a lot of ARM boards and SoCs. Platform-dependent code is in directories named `mach-*` in `arch/arm`, approximately one per SoC. There are other directories named `plat-*` which contain code common to several versions of an SoC. In the case of the Nova board, the relevant directory is `mach-omap2`. Don't be fooled by the name, though, it contains support for OMAP2, 3, and 4 chips.

In the following sections, I am going to do the port to the Nova board in two different ways. Firstly, I am going to show you how to do this with a device tree, and then without, since there are a lot of devices in the field that fit in this category. You will see that it is much simpler when you have a device tree.

With a device tree

The first thing to do is create a device tree for the board and modify it to describe the additional or changed hardware on the board. In this simple case, we will just copy `am335x-boneblack.dts` to `nova.dts` and change the board name:

```
/dts-v1/;
#include "am33xx.dtsci"
#include "am335x-bone-common.dtsci"
{
    model = "Nova";
    compatible = "ti,am335x-bone-black", "ti,am335x-bone", "ti,am33xx";
};

...
```

We can build `nova.dtb` explicitly:

```
$ make ARCH=arm nova.dtb
```

Or, if we want `nova.dtb` to be produced by default for the OMAP2 platform with `make ARCH=arm dtbs` then we could add the following line to `arch/arm/boot/dts/Makefile`:

```
dtb-$ (CONFIG_SOC_AM33XX) += \
[...]
nova.dtb \
[...]
```

Now we can boot the same `zImage` file as before, configured with `multi_v7_defconfig`, but load the `nova.dtb` as we can see here:

```
Starting kernel ...

[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Initializing cgroup subsys cpuset
[    0.000000] Initializing cgroup subsys cpu
[    0.000000] Initializing cgroup subsys cpuacct
[    0.000000] Linux version 3.18.3-dirty (chris@builder) (gcc
version 4.9.1 (crosstool-N
G 1.20.0) ) #1 SMP Wed Jan 28 07:50:50 GMT 2015
[    0.000000] CPU: ARMv7 Processor [413fc082] revision 2 (ARMv7),
cr=10c5387d
[    0.000000] CPU: PIPT / VIPT nonaliasing data cache, VIPT
aliasing instruction cache
[    0.000000] Machine model: Nova
...
```

We could create a custom configuration by taking a copy of `multi_v7_defconfig` and adding in those features we need, and cutting down code size by leaving out those we don't.

Without a device tree

Firstly, we need to create a configuration name for the board, in this case, it is NOVABOARD. We need to add this to the Kconfig file of the mach- directory for your SoC and we need to add a dependency for the SoC support itself, which is OMAPAM33XX.

These lines are added to arch/arm/mach-omap2/Kconfig:

```
config MACH_NOVA BOARD
    bool "Nova board"
    depends on SOC_OMAPAM33XX
    default n
```

There is a source file named board-* .c for each board, which contains code and configurations which are specific to the target. In our case, it is board-nova .c, based on a copy of board-am335xevm .c. There must be a rule to compile it, conditional on CONFIG_MACH_NOVABOARD, which this addition to arch/arm/mach-omap2/Makefile takes care of:

```
obj-$ (CONFIG_MACH_NOVABOARD) += board-nova.o
```

Since we are not using the device tree to identify the board, we will have to use the older machine number mechanism. This is a number unique to each board that is passed by the bootloader in register r1, and which the ARM start code will use to select the correct board support. The definitive list of ARM machine numbers is held at: www.arm.linux.org.uk/developer/machines/download.php. You can request a new machine number from: www.arm.linux.org.uk/developer/machines/?action=new#.

If we hijack machine number 4242, we could add it to arch/arm/tools/mach-types, as shown:

machine_is_xxx	CONFIG_xxxx	MACH_TYPE_xxx	number
...			
nova_board	MACH_NOVABOARD	NOVABOARD	4242

When we build the kernel, it will be used to create the mach-types.h header file present in include/generated/.

The machine number and the board support are tied together by a structure which is defined like this:

```
MACHINE_START(NOVABOARD, "nova_board")
/* Maintainer: Chris Simmonds */
.atag_offset = 0x100,
```

```
.map_io      = am335x_evm_map_io,
.init_early   = am33xx_init_early,
.init_irq     = ti81xx_init_irq,
.handle_irq   = omap3_intc_handle_irq,
.timer        = &omap3_am33xx_timer,
.init_machine = am335x_evm_init,
MACHINE_END
```

Note that there may be more than one machine structure in a board file, allowing us to create a kernel that will run on several different boards. The machine number passed by the bootloader selects the correct one.

Finally, we need a new default configuration for our board, which selects CONFIG_MACH_NOVABOARD and other configuration options specific to it. In the following example, it would be in arch/arm/configs/novaboard_defconfig. Now you can build the kernel image as usual:

```
$ make ARCH=arm novaboard_defconfig
$ make -j 4 ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabi-
zImage
```

There is still one step before the job is finished. The bootloader needs to be modified to pass the right machine number. Assuming that you are using U-Boot, you need to copy the machine numbers generated by Linux in arch/arm/include/asm/mach-types.h to U-Boot file arch/arm/include/asm/mach-types.h. Then you need to update the configuration header file for Nova, include/configs/nova.h, and add the following line:

```
#define CONFIG_MACH_TYPE           MACH_TYPE_NOVABOARD
```

Now, at last, you can build U-Boot and use it to boot the new kernel on the Nova board:

```
Starting kernel ...

[    0.000000] Linux version 3.2.0-00246-g0c74d7a-dirty
(chris@builder) (gcc version 4.9.
1 (crosstool-NG 1.20.0) ) #3 Wed Jan 28 11:45:10 GMT 2015
[    0.000000] CPU: ARMv7 Processor [413fc082] revision 2 (ARMv7),
cr=10c53c7d
[    0.000000] CPU: PIPT / VIPT nonaliasing data cache, VIPT
aliasing instruction cache
[    0.000000] Machine: nova_board
```

Additional reading

The following resources have further information about the topics introduced in this chapter:

- *Linux Kernel Newbies*, kernelnewbies.org
- *Linux Weekly News*, www.lwn.net

Summary

Linux is a very powerful and complex operating system kernel that can be married to various types of user space ranging from a simple embedded device, to increasingly complex mobile devices using Android, to a full server operating system. One of its strengths is the degree of configurability. The definitive place to get the source code is www.kernel.org, but you will probably need to get the source for a particular SoC or board from the vendor of that device or a third-party that supports that device. The customization of the kernel for a particular target may consist of changes to the core kernel code, additional drivers for devices that are not in mainline Linux, a default kernel configuration file and, a device tree source file.

Normally you start with the default configuration for your target board, and then tweak it by running one of the configuration tools such as `menuconfig`. One of the things you should consider at this point is whether kernel features and drivers should be compiled as modules or built-in. Kernel modules are usually no great advantage for embedded systems, where the feature set and hardware are usually well defined. However, modules are often used as a way to import proprietary code into the kernel, and also to reduce boot time by loading non-essential drivers after boot. Building the kernel produces a compressed kernel image file, named `zImage`, `bzImage`, or `uImage` depending on the bootloader you will be using and the target architecture. A kernel build will also generate any kernel modules (as `.ko` files) that you have configured, and device tree binaries (as `.dtb` files) if your target requires them.

Porting Linux to a new target board can be quite simple or very difficult depending on how different the hardware is from that in the mainline or vendor supplied kernel. If your hardware is based on a well-known reference design, then it may be just a question of making changes to the device tree or to the platform data. You may well need to add device drivers, which I discuss in *Chapter 8, Introducing Device Drivers*. However, if the hardware is radically different to a reference design, you may need additional core support, which is outside the scope of this book.

The kernel is the core of a Linux based system, but it cannot work by itself. It requires a root filesystem that contains user space. The root filesystem can be a ramdisk or a filesystem accessed via a block device, which will be the subject of the next chapter. As we have seen, booting a kernel without a root filesystem results in a kernel panic.

5

Building a Root Filesystem

The root filesystem is the fourth and final element of embedded Linux. Once you have read this chapter, you will be able build, boot, and run a simple embedded Linux system.

This chapter explores the fundamental concepts behind the root filesystem by building one from scratch. The main aim is to provide the background information that you need to understand and make best use of build systems like Buildroot and the Yocto Project, which I will cover in *Chapter 6, Selecting a Build System*.

The techniques I will describe here are broadly known as **roll your own** or **RYO**. Back in the earlier days of embedded Linux, it was the only way to create a root filesystem. There are still some use cases where an RYO root filesystem is applicable, for example, when the amount of RAM or storage is very limited, for quick demonstrations, or for any case in which your requirements are not (easily) covered by the standard build system tools. Nevertheless, these cases are quite rare. Let me emphasize that the purpose of this chapter is educational, it is not meant to be a recipe for building everyday embedded systems: use the tools described in the next chapter for that.

The first objective is to create a minimal root filesystem that will give us a shell prompt. Then, using that as a base, we will add scripts to start up other programs and configure a network interface and user permissions. Knowing how to build the root filesystem from scratch is a useful skill and it will help you to understand what is going on when we look at more complex examples in later chapters.

What should be in the root filesystem?

The kernel will get a root filesystem, either as a ramdisk, passed as a pointer from the bootloader, or by mounting the block device given on the kernel command line by the `root=` parameter. Once it has a root filesystem, the kernel will execute the first program, by default named `init`, as described in the section *Early Userspace* in *Chapter 4, Porting and Configuring the Kernel*. Then, as far as the kernel is concerned, its job is complete. It is up to the `init` program to begin processing scripts, start other programs, and so on, by calling system functions in the C library, which translate into kernel system calls.

To make a useful system, you need these components as a minimum:

- **init**: The program that starts everything off, usually by running a series of scripts.
- **shell**: Needed to give you a command prompt but, more importantly, to run the shell scripts called by `init` and other programs.
- **daemons**: Various server programs, started by `init`.
- **libraries**: Usually, the programs mentioned so far are linked with shared libraries which must be present in the root filesystem.
- **Configuration files**: The configuration for `init` and other daemons is stored in a series of ASCII text files, usually in the `/etc` directory.
- **Device nodes**: The special files that give access to various device drivers.
- **/proc and /sys**: Two pseudo filesystems that represent kernel data structures as a hierarchy of directories and files. Many programs and library functions read these files.
- **kernel modules**: If you have configured some parts of your kernel to be modules, they will be here, usually in `/lib/modules/[kernel version]`.

In addition, there are the system application or applications that make the device do the job it is intended for, and the runtime end user data that they collect.

As an aside, it is possible to condense all of the above into a single program. You could create a statically linked program that is started instead of `init` and runs no others. I have come across such a configuration only once. For example, if your program was named `/myprog`, you would put the following command in the kernel command line:

```
init=/myprog
```

Or, if the root filesystem was loaded as a ramdisk, you would put the following command:

```
rdinit=/myprog
```

The downside of this approach is that you can't make use of the many tools that normally go into an embedded system; you have to do everything yourself.

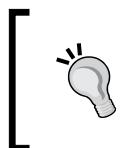
Directory layout

Interestingly, Linux does not care about the layout of files and directories beyond the existence of the program named by `init=` or `rdinit=`, so you are free to put things wherever you like. As an example, compare the file layout of a device running Android to that of a desktop Linux distribution: they are almost completely different.

However, many programs expect certain files to be in certain places, and it helps us developers if devices use a similar layout, Android aside. The basic layout of a Linux system is defined in the **Filesystem Hierarchy Standard (FHS)**, see the reference at the end of this chapter. The FHS covers all implementations of Linux operating systems from the largest to the smallest. Embedded devices have a sub-set based on need but it usually includes the following:

- `/bin`: programs essential for all users
- `/dev`: device nodes and other special files
- `/etc`: system configuration
- `/lib`: essential shared libraries, for example, those that make up the C library
- `/proc`: the `proc` filesystem
- `/sbin`: programs essential to the system administrator
- `/sys`: the `sysfs` filesystem
- `/tmp`: a place to put temporary or volatile files
- `/usr`: as a minimum, this should contain the directories `/usr/bin`, `/usr/lib` and `/usr/sbin`, which contain additional programs, libraries, and system administrator utilities
- `/var`: a hierarchy of files and directories that may be modified at runtime, for example, log messages, some of which must be retained after boot

There are some subtle distinctions here. The difference between `/bin` and `/sbin` is simply that `/sbin` need not be included in the search path for non-root users. Users of Red Hat-derived distributions will be familiar with this. The significance of `/usr` is that it may be in a separate partition from the root filesystem so it cannot contain anything that is needed to boot the system up. That is what essential means in the preceding description: it contains files that are needed at boot time and so must be part of the root filesystem.



While it might seem like overkill to have four directories to store programs, a counter argument would be that it does no harm, and it may even do some good because it allows you to store `/usr` in a different filesystem.



Staging directory

You should begin by creating a staging directory on your host computer where you can assemble the files that will eventually be transferred to the target. In the following examples, I have used `~/rootfs`. You need to create a skeleton directory structure in that, for example:

```
$ mkdir ~/rootfs
$ cd ~/rootfs
$ mkdir bin dev etc home lib proc sbin sys tmp usr var
$ mkdir usr/bin usr/lib usr/sbin
$ mkdir var/log
```

To see the directory hierarchy more clearly you can use the handy `tree` command, used in the following example with the `-d` option to show only directories:

```
$ tree -d
```

```
└── bin
└── dev
└── etc
└── home
└── lib
└── proc
└── sbin
└── sys
```

```
└── tmp
└── usr
|   ├── bin
|   ├── lib
|   └── sbin
└── var
    └── log
```

POSIX file access permissions

Every process which, in the context of this discussion, means every running program, belongs to a user and one or more groups. The user is represented by a 32-bit number called the **user ID** or **UID**. Information about users, including the mapping from a UID to a name, is kept in `/etc/passwd`. Likewise, groups are represented by a **group ID** or **GID**, with information kept in `/etc/group`. There is always a root user with a UID of 0 and a root group with a GID of 0. The root user is also called the super-user because, in a default configuration, it bypasses most permission checks and can access all the resources in the system. Security in Linux-based systems is mainly about restricting access to the root account.

Each file and directory also has an owner and belongs to exactly one group. The level of access a process has to a file or directory is controlled by a set of access permission flags, called the mode of the file. There are three collections of three bits: the first collection applies to the owner of the file, the second to members of the same group as the file, and the last to everyone else, the rest of the world. The bits are for read (r), write (w), and execute (x) permissions on the file. Since three bits fit neatly into an octal digit, they are usually represented in octal, as shown in the following figure:

400	r-----	Owner permissions
200	-w-----	
100	--x-----	
040	---r----	Group permissions
020	----w---	
010	----x--	
004	-----r-	World permissions
002	-----w-	
001	-----x	

There is a further group of three bits that have special meanings:

- **SUID (4)**: If the file is an executable, change the effective UID of the process to that of the owner of the file.
- **SGID (2)**: If the file is an executable, change the effective GID of the process to that of the group of the file.
- **Sticky (1)**: In a directory, restrict deletion so that one user cannot delete files that are owned by another user. This is usually set on `/tmp` and `/var/tmp`.

The SUID bit is probably the most often used. It gives non-root users a temporary privilege escalation to super-user to perform a task. A good example is the `ping` program: `ping` opens a raw socket which is a privileged operation. In order for normal users to use `ping`, it is normally owned by the root and has the SUID bit set so that, when you run `ping`, it executes with UID 0 regardless of your UID.

To set these bits, use the octal numbers, 4, 2, 1, with the `chmod` command. For example, to set SUID on `/bin/ping` in your staging root directory, you could use the following:

```
$ cd ~/rootfs  
$ ls -l bin/ping  
-rwxr-xr-x 1 root root 35712 Feb 6 09:15 bin/ping  
$ sudo chmod 4755 bin/ping  
$ ls -l bin/ping  
-rwsr-xr-x 1 root root 35712 Feb 6 09:15 bin/ping
```



Note the `s` in the last file listing: that is the indication that SUID is set.



File ownership permissions in the staging directory

For security and stability reasons, it is vitally important to pay attention to the ownership and permissions of the files that will be placed on the target device. Generally speaking, you want to restrict sensitive resources to be accessible only by the root and to run as many of the programs using non-root users so that, if they are compromised by an outside attack, they offer as few system resources to the attacker as possible. For example, the device node `/dev/mem` gives access to system memory, which is necessary in some programs. But, if it is readable and writeable by everyone, then there is no security because everyone can access everything. So `/dev/mem` should be owned by root, belong to the root group and have a mode of 600, which denies read and write access to all but the owner.

There is a problem with the staging directory though. The files you create there will be owned by you but, when they are installed on the device, they should belong to specific owners and groups, mostly the root user. An obvious fix is to change the ownership at this stage with the command shown here:

```
$ cd ~/rootfs  
$ sudo chown -R root:root *
```

The problem is that you need root privileges to run that command and, from that point onward, you will need to be root to modify any files in the staging directory. Before you know it, you are doing all your development logged on as root, which is not a good idea. This is a problem that we will come back to later.

Programs for the root filesystem

Now, it is time to start populating the root filesystem with the essential programs and the supporting libraries, configuration, and data files that it needs to operate, beginning with an overview of the types of program you will need.

The init program

You have seen in the previous chapter that `init` is the first program to be run and so has PID 1. It runs as the root user and so has maximum access to system resources. Usually, it runs shell scripts which start daemons: a daemon is a program that runs in the background with no connection to a terminal, in other places it would be called a server program.

Shell

We need a shell to run scripts and to give us a command-line prompt so that we can interact with the system. An interactive shell is probably not necessary in a production device, but it is useful for development, debugging, and maintenance. There are various shells in common use in embedded systems:

- `bash`: is the big beast that we all know and love from desktop Linux. It is a superset of the Unix Bourne shell, with many extensions or *bashisms*.
- `ash`: also based on the Bourne shell, and has a long history with the BSD variants of Unix. Busybox has a version of `ash` which has been extended to make it more compatible with `bash`. It is much smaller than `bash` and hence is a very popular choice for embedded systems.

- `hush`: is a very small shell that we briefly looked at in the chapter on bootloaders. It is useful on devices with very little memory. There is a version in BusyBox.

 If you are using `ash` or `hush` as the shell on the target, make sure that you test your shell scripts on the target. It is very tempting to test them only on the host, using `bash`, and then be surprised that they don't work when you copy them to the target.

Utilities

The shell is just a way of launching other programs and a shell script is little more than a list of programs to run, with some flow control and a means of passing information between programs. To make a shell useful, you need the utility programs that the Unix command-line is based on. Even for a basic root filesystem, there are approximately 50 utilities, which presents two problems. Firstly, tracking down the source code for each and cross compiling it would be quite a big job. Secondly, the resulting collection of programs would take up several tens of megabytes, which was a real problem in the early days of embedded Linux when a few megabytes was all you had. To solve this problem, BusyBox was born.

BusyBox to the rescue!

The genesis of BusyBox had nothing to do with embedded Linux. The project was instigated in 1996 by Bruce Perens for the Debian installer so that he could boot Linux from a 1.44 MB floppy disk. Coincidentally, that was about the size of the storage on contemporary devices and so the embedded Linux community quickly took it up. BusyBox has been at the heart of embedded Linux ever since.

BusyBox was written from scratch to perform the essential functions of those essential Linux utilities. The developers took advantage of the 80:20 rule: the most useful 80% of a program is implemented in 20% of the code. Hence, BusyBox tools implement a subset of the functions of the desktop equivalents, but they do enough to be useful in the majority of cases.

Another trick BusyBox employs is to combine all the tools together into a single binary, making it easy to share code between them. It works like this: BusyBox is a collection of applets, each of which exports its main function in the form `[applet]_main`. For example, the `cat` command is implemented in `coreutils/cat.c` and exports `cat_main`. The main function of BusyBox itself dispatches the call to the correct applet based on the command-line arguments.

So, to read a file, you can launch `busybox` with the name of the applet you want to run, followed by any arguments the applet expects, as shown here:

```
$ busybox cat my_file.txt
```

You can also run `busybox` with no arguments to get a list of all the applets that have been compiled.

Using BusyBox in this way is rather clumsy. A better way to get BusyBox to run the `cat` applet is to create a symbolic link from `/bin/cat` to `/bin/busybox`:

```
$ ls -l bin/cat bin/busybox
-rwxr-xr-x 1 chris chris 892868 Feb  2 11:01 bin/busybox
lrwxrwxrwx 1 chris chris      7 Feb  2 11:01 bin/cat -> busybox
```

When you type `cat` at the command line, `busybox` is the program that actually runs. BusyBox only has to check the command tail passed in `argv[0]`, which will be `/bin/cat`, extract the application name, `cat`, and do a table look-up to match `cat` with `cat_main`. All this is in `libbb/appletlib.c` in this section of code (slightly simplified):

```
applet_name = argv[0];
applet_name = bb_basename(applet_name);
run_applet_and_exit(applet_name, argv);
```

BusyBox has over three hundred applets including an `init` program, several shells of varying levels of complexity, and utilities for most admin tasks. There is even a simple version of the `vi` editor so you can change text files on your device.

To summarize, a typical installation of BusyBox consists of a single program with a symbolic link for each applet, but which behaves exactly as if it were a collection of individual applications.

Building BusyBox

BusyBox uses the same Kconfig and Kbuild system of the kernel, so cross compiling is straightforward. You can get the source by cloning the git archive and checking out the version you want (`1_24_1` was the latest at the time of writing), like this:

```
$ git clone git://busybox.net/busybox.git
$ cd busybox
$ git checkout 1_24_1
```

You can also download the corresponding tarball file from <http://busybox.net/downloads>. Then, configure BusyBox, starting in this case with the default configuration, which enables pretty much all of the features of BusyBox:

```
$ make distclean  
$ make defconfig
```

At this point, you probably want to run `make menuconfig` to fine tune the configuration. You almost certainly want to set the install path in **Busybox Settings | Installation Options** (`CONFIG_PREFIX`) to point to the staging directory. Then, you can cross compile in the usual way:

```
$ make -j 4 ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf-
```

The result is the executable, `busybox`. For a `defconfig` build for ARM v7a, it comes out at about 900 KiB. If that is too big for you, you can slim it down by configuring out the utilities you don't need.

To install BusyBox, use the following command:

```
$ make install
```

This will copy the binary to the directory configured in `CONFIG_PREFIX` and create all the symbolic links to it.

ToyBox – an alternative to BusyBox

BusyBox is not the only game in town. For example, Android has an equivalent named Toolbox, but it is more tuned to the needs of Android and not useful in a general purpose embedded environment. A more useful option is ToyBox, a project started and maintained by Rob Landley, who was previously a maintainer of BusyBox. ToyBox has the same aim as BusyBox, but with more emphasis on complying with standards, especially POSIX-2008 and LSB 4.1, and less on compatibility with GNU extensions to those standards. ToyBox is smaller than BusyBox, partly because it implements fewer applets.

However, the main difference is the license, BSD rather than GPL v2, which makes it license-compatible with operating systems with a BSD-licensed user space, such as Android itself.

Libraries for the root filesystem

Programs are linked with libraries. You could link them all statically, in which case, there would be no libraries on the target device. But, that takes up an unnecessarily large amount of storage if you have more than two or three programs. So, you need to copy shared libraries from the toolchain to the staging directory. How do you know which libraries?

One option is to copy all of them since they must be of some use, otherwise they wouldn't exist! That is certainly logical and, if you are creating a platform to be used by others for a range of applications, that would be the correct approach. Be aware, though, that a full glibc is quite large. In the case of a CrossTool-NG build of glibc 2.19, the space taken by /lib and /usr/lib is 33 MiB. Of course, you could cut down on that considerably by using uClibc or musl libc libraries.

Another option is to cherry pick only those libraries that you require, for which you need a means of discovering library dependencies. Using some of our knowledge from *Chapter 2, Learning About Toolchains* libraries, you can use `readelf` for that task:

```
$ cd ~/rootfs
$ arm-cortex_a8-linux-gnueabihf-readelf -a bin/busybox | grep "program
interpreter"
[Requesting program interpreter: /lib/ld-linux-armhf.so.3]
$ arm-cortex_a8-linux-gnueabihf-readelf -a bin/busybox | grep "Shared
library"
0x00000001 (NEEDED)           Shared library: [libm.so.6]
0x00000001 (NEEDED)           Shared library: [libc.so.6]
```

Now you need to find these files in the toolchain and copy them to the staging directory. Remember that you can find `sysroot` like this:

```
$ arm-cortex_a8-linux-gnueabihf-gcc -print-sysroot
/home/chris/x-tools/arm-cortex_a8-linux-gnueabihf/arm-cortex_a8-
linux-gnueabihf/sysroot
```

To reduce the amount of typing, I am going to keep a copy of that in a shell variable:

```
$ export SYSROOT=`arm-cortex_a8-linux-gnueabihf-gcc -print-
sysroot`
```

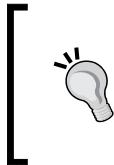
If you look at `/lib/ld-linux-armhf.so.3`, in `sysroot`, you will see that, it is, in fact, a symbolic link:

```
$ ls -l $SYSROOT/lib/ld-linux-armhf.so.3
[...]/sysroot/lib/ld-linux-armhf.so.3 -> ld-2.19.so
```

Repeat the exercise for `libc.so.6` and `libm.so.6` and you will end up with a list of three files and three symbolic links. Copy them using `cp -a`, which will preserve the symbolic link:

```
$ cd ~/rootfs
$ cp -a $SYSROOT/lib/ld-linux-armhf.so.3 lib
$ cp -a $SYSROOT/lib/ld-2.19.so lib
$ cp -a $SYSROOT/lib/libc.so.6 lib
$ cp -a $SYSROOT/lib/libc-2.19.so lib
$ cp -a $SYSROOT/lib/libm.so.6 lib
$ cp -a $SYSROOT/lib/libm-2.19.so lib
```

Repeat this procedure for each program.



[It is only worth doing this to get the very smallest embedded footprint possible. There is a danger that you will miss libraries that are loaded through `dlopen(3)` calls - plugins mostly. We will look at an example with the NSS libraries when we come to configure network interfaces later on in this chapter.]

Reducing size by stripping

Libraries and programs are often compiled with a symbol table information built in, more so if you have compiled with the debug switch, `-g`. You seldom need these on the target. A quick and easy way to save space is to strip them. This example shows `libc` before and after stripping:

```
$ file rootfs/lib/libc-2.19.so
rootfs/lib/libc-2.19.so: ELF 32-bit LSB shared object, ARM, version 1
(SYSV), dynamically linked (uses shared libs), for GNU/Linux 3.15.4,
not stripped
$ ls -og rootfs/lib/libc-2.19.so
-rwxrwxr-x 1 1547371 Feb 5 10:18 rootfs/lib/libc-2.19.so
$ arm-cortex_a8-linux-gnueabi-strip rootfs/lib/libc-2.19.so
$ file rootfs/lib/libc-2.19.so
rootfs/lib/libc-2.19.so: ELF 32-bit LSB shared object, ARM, version 1
(SYSV), dynamically linked (uses shared libs), for GNU/Linux 3.15.4,
stripped
$ ls -l rootfs/lib/libc-2.19.so
-rwxrwxr-x 1 chris chris 1226024 Feb 5 10:19 rootfs/lib/libc-2.19.so
$ ls -og rootfs/lib/libc-2.19.so
-rwxrwxr-x 1 1226024 Feb 5 10:19 rootfs/lib/libc-2.19.so
```

In this case, we saved 321,347 bytes, which was about 20%.

When stripping kernel modules, use the following command:

```
strip --strip-unneeded <module name>
```

Otherwise, you will strip out the symbols needed to relocate the module code and it will fail to load.

Device nodes

Most devices in Linux are represented by device nodes, in accordance with the Unix philosophy that *everything is a file* (except network interfaces, which are sockets). A device node may refer to a block device or a character device. Block devices are mass storage devices such as SD cards or hard drives. A character device is pretty much anything else, once again with the exception of network interfaces. The conventional location for device nodes is the directory `/dev`. For example, a serial port may be represented by the device node `/dev/ttys0`.

Device nodes are created using the program `mknod` (short for make node):

```
mknod <name> <type> <major> <minor>
```

`name` is the name of the device node that you want to create, `type` is either, `c` for character devices, and `b` for block. They each have a major number and a minor number which is used by the kernel to route file requests to the appropriate device driver code. There is a list of standard major and minor numbers in the kernel source in `Documentation/devices.txt`.

You will need to create device nodes for all the devices you want to access on your system. You can do that manually by using the `mknod` command as I will illustrate here, or you can use one of the device managers mentioned later to create them automatically, at runtime.

You need just two nodes to boot with BusyBox: `console` and `null`. The `console` only needs to be accessible to root, the owner of the device node, so the access permissions are 600. The `null` device should be readable and writable by everyone, so the mode is 666. You can use the `-m` option to `mknod` to set the mode when creating the node. You need to be root to create a device node:

```
$ cd ~/rootfs
$ sudo mknod -m 666 dev/null c 1 3
$ sudo mknod -m 600 dev/console c 5 1
$ ls -l dev
total 0
crw----- 1 root root 5, 1 Oct 28 11:37 console
crw-rw-rw- 1 root root 1, 3 Oct 28 11:37 null
```

You delete device nodes by using the standard `rm` command: there is no `rmmnod` command because, once created, they are just files.

The proc and sysfs filesystems

proc and sysfs are two pseudo filesystems that give a window onto the inner workings of the kernel. They both represent kernel data as files in a hierarchy of directories: when you read one of the files, the contents you see do not come from disk storage, it has been formatted on-the-fly by a function in the kernel. Some files are also writable, meaning that a kernel function is called with the new data you have written and, if it is of the correct format and you have sufficient permissions, it will modify the value stored in the kernel's memory. In other words, proc and sysfs provide another way to interact with device drivers and other kernel code.

proc and sysfs should be mounted on the directories /proc and /sys:

```
mount -t proc proc /proc
mount -t sysfs sysfs /sys
```

Although they are very similar in concept, they perform different functions. proc has been part of Linux since the early days. Its original purpose was to expose information about processes to user space, hence the name. To this end, there is a directory for each process named /proc/<PID> which contains information about its state. The process list command, ps, reads these files to generate its output. In addition, there are files that give information about other parts of the kernel, for example, /proc/cpuinfo tells you about the CPU, /proc/interrupts has information about interrupts, and so on. Finally, in /proc/sys, there are files that display and control the state and behavior of kernel sub-systems, especially scheduling, memory management, and networking. The best reference for the files you will find in proc is man page proc (5).

In fact, over time, the number of files in proc and their layout has become rather chaotic. In Linux 2.6, sysfs was introduced to export a subset of the data in an ordered way.

In contrast, sysfs exports a very ordered hierarchy of files relating to devices and the way they are connected to each other.

Mounting filesystems

The mount command allows us to attach one filesystem to a directory within another, forming a hierarchy of filesystems. The one at the top, which was mounted by the kernel when it booted, is called the root filesystem. The format of the mount command is as follows:

```
mount [-t vfstype] [-o options] device directory
```

You need to specify the type of the filesystem, `vfstype`, the block device node it resides on, and the directory you want to mount it to. There are various options you can give after the `-o`, have a look at the manual for more information. As an example, if you want to mount an SD card containing an `ext4` filesystem in the first partition onto directory `/mnt`, you would type the following:

```
mount -t ext4 /dev/mmcblk0p1 /mnt
```

Assuming the mount succeeds, you would be able to see the files stored on the SD card in the directory `/mnt`. In some cases, you can leave out the filesystem type and let the kernel probe the device to find out what is stored there.

Looking at the example of mounting the `proc` filesystem, there is something odd: there is no device node, `/dev/proc`, since it is a pseudo filesystem, not a real one. But the `mount` command requires a device as a parameter. Consequently we have to give a string where the device should go, but it does not matter much what that string is. These two commands achieve exactly the same result:

```
mount -t proc proc /proc
mount -t proc nodevice /proc
```

It is fairly common to use the filesystem type in the place of the device when mounting pseudo filesystems.

Kernel modules

If you have kernel modules, they need to be installed into the root filesystem, using the kernel `make modules_install` target, as we saw in the last chapter. This will copy them into the directory `/lib/modules/<kernel version>` together with the configuration files needed by the `modprobe` command.

Be aware that you have just created a dependency between the kernel and the root filesystem. If you update one, you will have to update the other.

Transferring the root filesystem to the target

Having created a skeleton root filesystem in your staging directory, the next task is to transfer it to the target. In the sections that follow, I will describe three possibilities:

- **ramdisk**: a filesystem image that is loaded into RAM by the bootloader. Ramdisks are easy to create and have no dependencies on mass storage drivers. They can be used in fall-back maintenance mode when the main root filesystem needs updating. They can even be used as the main root filesystem in small embedded devices and, of course, as the early user space in mainstream Linux distributions. A compressed ramdisk uses the minimum amount of storage but still consumes RAM. The contents are volatile so you need another storage type to store permanent data such as configuration parameters.
- **disk image**: a copy of the root filesystem formatted and ready to be loaded onto a mass storage device on the target. For example, it could be an image in ext4 format ready to be copied onto an SD card, or it could be in jffs2 format ready to be loaded into flash memory via the bootloader. Creating a disk image is probably the most common option. There is more information about the different types of mass storage in *Chapter 7, Creating a Storage Strategy*.
- **network filesystem**: the staging directory can be exported to the network via an NFS server and mounted by the target at boot-time. This is often done during the development phase in preference to repeated cycles of creating a disk image and reloading it onto the mass storage device, which is quite a slow process.

I will start with ramdisk and use it to illustrate a few refinements to the root filesystem, like adding user names and a device manager to create device nodes automatically. Then, I will show you how to create a disk image and, finally, how to use NFS to mount the root filesystem over a network.

Creating a boot ramdisk

A Linux boot ramdisk, strictly speaking, an **initial RAM filesystem** or **initramfs**, is a compressed cpio archive. cpio is an old Unix archive format, similar to TAR and ZIP but easier to decode and so requiring less code in the kernel. You need to configure your kernel with `CONFIG_BLK_DEV_INITRD` to support initramfs.

In fact, there are three different ways to create a boot ramdisk: as a standalone `cpio` archive, as a `cpio` archive embedded in the kernel image, and as a device table which the kernel build system processes as part of the build. The first option gives the most flexibility because we can mix and match kernels and ramdisks to our hearts content. However, it means that you have two files to deal with instead of one and not all bootloaders have the facility to load a separate ramdisk. I will show you how to build one into the kernel later.

Standalone ramdisk

The following sequence of instructions creates the archive, compresses it and adds a U-Boot header ready for loading onto the target:

```
$ cd ~/rootfs
$ find . | cpio -H newc -ov --owner root:root > ../initramfs.cpio
$ cd ..
$ gzip initramfs.cpio
$ mkimage -A arm -O linux -T ramdisk -d initramfs.cpio.gz uRamdisk
```

Note that we ran `cPIO` with the option `--owner root:root`. This is a quick fix for the file ownership problem mentioned earlier, making everything in the `cPIO` file UID and GID 0.

The final size of the `uRamdisk` file is ~ 2.9 MiB, with no kernel modules. Add to that 4.4 MiB for the kernel `zImage` file, and 440 KiB for U-Boot and this gives a total of 7.7 MiB of storage needed to boot this board. We are a little way off the 1.44 MiB floppy that started it all off. If size was a real problem, you could use one of these options:

- Make the kernel smaller by leaving out drivers and functions you don't need
- Make BusyBox smaller by leaving out utilities you don't need
- Use uClibc or musl libc in place of glibc
- Compile BusyBox statically

Booting the ramdisk

The simplest thing we can do is to run a shell on the console so that we can interact with the device. We can do that by adding `rdinit=/bin/sh` to the kernel command line. Now, you can boot the device.

Booting with QEMU

QEMU has the option `-initrd` to load `initramfs` into memory, so the full command is now as follows:

```
$ cd ~/rootfs
$ QEMU_AUDIO_DRV=none \
qemu-system-arm -m 256M -nographic -M vexpress-a9 -kernel zImage
-append "console=ttyAMA0 rdinit=/bin/sh" -dtb vexpress-v2p-ca9.dtb
-initrd initramfs.cpio.gz
```

Booting the BeagleBone Black

To boot the BeagleBone Black, boot to the U-Boot prompt and enter these commands:

```
fatload mmc 0:1 0x80200000 zImage
fatload mmc 0:1 0x80f00000 am335x-boneblack.dtb
fatload mmc 0:1 0x81000000 uRamdisk
setenv bootargs console=ttyO0,115200 rdinit=/bin/sh
bootz 0x80200000 0x81000000 0x80f00000
```

If all goes well, you will get a root shell prompt on the console.

Mounting proc

Note that the `ps` command doesn't work: that is because the `proc` filesystem has not been mounted yet. Try mounting it and run `ps` again.

A refinement to this setup is to write a shell script that contains things that need to be done at boot-up and give that as the parameter to `rdinit=`. The script would look like the following snippet:

```
#!/bin/sh
/bin/mount -t proc proc /proc
/bin/sh
```

Using a shell as `init` in this way is very handy for quick hacks, for example, when you want to rescue a system with a broken `init` program. However, in most cases, you would use an `init` program, which we will cover further down.

Building a ramdisk cpio into the kernel image

In some cases, it is preferable to build the ramdisk into the kernel image, for example, if the bootloader cannot handle a ramdisk file. To do this, change the kernel configuration and set `CONFIG_INITRAMFS_SOURCE` to the full path of the `cpio` archive you created earlier. If you are using `menuconfig`, it is in **General setup | Initramfs source file(s)**. Note that it has to be the uncompressed `cpio` file ending in `.cpio`; not the gzipped version. Then, build the kernel. You should see that it is larger than before.

Booting is the same as before, except that there is no ramdisk file. For QEMU, the command is like this:

```
$ cd ~/rootfs
$ QEMU_AUDIO_DRV=none \
qemu-system-arm -m 256M -nographic -M vexpress-a9 -kernel zImage -
append "console=ttyAMA0 rdinit=/bin/sh" -dtb vexpress-v2p-ca9.dtb
```

For the BeagleBone Black, enter these commands into U-Boot:

```
fatload mmc 0:1 0x80200000 zImage
fatload mmc 0:1 0x80f00000 am335x-boneblack.dtb
setenv bootargs console=ttyO0,115200 rdinit=/bin/sh
bootz 0x80200000 - 0x80f00000
```

Of course, you must remember to rebuild the kernel each time you change the contents of the ramdisk and regenerate the `.cpio` file.

Another way to build a kernel with ramdisk

An interesting way to build the ramdisk into the kernel image is by using a **device table** to generate a `cpio` archive. A device table is a text file which lists the files, directories, device nodes, and links that go into the archive. The overwhelming advantage is that you can create entries in the `cpio` file that are owned by root, or any other UID, without having root privileges yourself. You can even create device nodes. All this is possible because the archive is just a data file. It is only when it is expanded by Linux at boot time that real files and directories get created, using the attributes you have specified.

Here is a device table for our simple `rootfs`, but missing most of the symbolic links to `busybox` to make it manageable:

```
dir /proc 0755 0 0
dir /sys 0755 0 0
```

```
dir /dev 0755 0 0
nod /dev/console 0600 0 0 c 5 1
nod /dev/null 0666 0 0 c 1 3
nod /dev/tty00 0600 0 0 c 252 0
dir /bin 0755 0 0
file /bin/busybox /home/chris/rootfs/bin/busybox 0755 0 0
slink /bin/sh /bin/busybox 0777 0 0
dir /lib 0755 0 0
file /lib/ld-2.19.so /home/chris/rootfs/lib/ld-2.19.so 0755 0 0
slink /lib/ld-linux.so.3 /lib/ld-2.19.so 0777 0 0
file /lib/libc-2.19.so /home/chris/rootfs/lib/libc-2.19.so 0755 0 0
slink /lib/libc.so.6 /lib/libc-2.19.so 0777 0 0
file /lib/libm-2.19.so /home/chris/rootfs/lib/libm-2.19.so 0755 0 0
slink /lib/libm.so.6 /lib/libm-2.19.so 0777 0 0
```

The syntax is fairly obvious:

- dir <name> <mode> <uid> <gid>
- file <name> <location> <mode> <uid> <gid>
- nod <name> <mode> <uid> <gid> <dev_type> <maj> <min>
- slink <name> <target> <mode> <uid> <gid>

The kernel provides a tool that reads this file and creates a cpio archive. The source is in `usr/gen_init_cpio.c`. There is a handy script in `scripts/gen_initramfs_list.sh` that creates a device table from a given directory, which saves a lot of typing.

To complete, the task, you need to set `CONFIG_INITRAMFS_SOURCE` to point to the device table file and then build the kernel. Everything else is the same as before.

The old initrd format

There is an older format for a Linux ramdisk, known as `initrd`. It was the only format available before Linux 2.6 and is still needed if you are using the mmu-less variant of Linux, uCLinux. It is pretty obscure and I will not cover it here. There is more information in the kernel source, in `Documentation/initrd.txt`.

The init program

Running a shell, or even a shell script, at boot time is fine for simple cases, but really you need something more flexible. Normally, Unix systems run a program called `init` that starts up and monitors other programs. Over the years, there have been many `init` programs, some of which I will describe in *Chapter 9, Starting up - the init Program*. For now, I will briefly introduce the `init` from BusyBox.

`init` begins by reading the configuration file, `/etc/inittab`. Here is a simple example which is adequate for our needs:

```
::sysinit:/etc/init.d/rcS
::askfirst:-/bin/ash
```

The first line runs a shell script, `rcS`, when `init` is started. The second line prints the message **Please press Enter to activate this console** to the console, and starts a shell when you press *Enter*. The leading `-` before `/bin/ash` means that it will be a login shell, which sources `/etc/profile` and `$HOME/.profile` before giving the shell prompt. One of the advantages of launching the shell like this is that job control is enabled. The most immediate effect is that you can use *Ctrl + C* to terminate the current program. Maybe you didn't notice it before but, wait until you run the `ping` program and find you can't stop it!

BusyBox `init` provides a default `inittab` if none is present in the root filesystem. It is a little more extensive than the preceding one.

The script `/etc/init.d/rcS` is the place to put initialization commands that need to be performed at boot, for example, mounting the `proc` and `sysfs` filesystems:

```
#!/bin/sh
mount -t proc proc /proc
mount -t sysfs sysfs /sys
```

Make sure that you make `rcS` executable, like this:

```
$ cd ~/rootfs
$ chmod +x etc/init.d/rcS
```

You can try it out on QEMU by changing the `-append` parameter, like this:

```
-append "console=ttyAMA0 rdinit=/sbin/init"
```

To achieve the same on the BeagleBone Black, you need to change the `bootargs` variable in U-Boot as shown:

```
setenv bootargs console=ttyO0,115200 rdinit=/sbin/init
```

Configuring user accounts

As I have hinted already, it is not good practice to run all programs as root since, if one is compromised by an outside attack, then the whole system is at risk and a misbehaving program can do more damage if it is running as root. It is preferable to create unprivileged user accounts and use them where full root is not necessary.

User names are configured in `/etc/passwd`. There is one line per user, with seven fields of information separated by colons:

- The login name
- A hash code used to verify the password , or more usually an x to indicate that the password is stored in `/etc/shadow`
- UID
- GID
- A comment field, often left blank
- The user's home directory
- (Optional) the shell this user will use

For example, this creates users `root` with UID 0 and `daemon` with UID 1:

```
root:x:0:0:root:/root:/bin/sh
daemon:x:1:1:daemon:/usr/sbin:/bin/false
```

Setting the shell for user `daemon` to `/bin/false` ensures that any attempt to log on with that name will fail.



Various programs have to read `/etc/passwd` so as to be able to look up UIDs and names, and so it has to be word-readable. That is a problem if the password hashes are stored in there because a malicious program would be able to take a copy and discover the actual passwords using a variety of cracker programs. Therefore, to reduce the exposure of this sensitive information, the passwords are stored in `/etc/shadow` and an x is placed in the password field to indicate that this is the case. `/etc/shadow` is only accessible as `root`, and, so as long as the `root` user is restricted, the passwords are safe.

The shadow password file consists of one entry per user, made up of nine fields. Here is an example that mirrors the `passwd` file shown in the preceding paragraph:

```
root::10933:0:99999:7:::
daemon:*:10933:0:99999:7:::
```

The first two fields are the username and the password hash. The remaining seven are related to password aging, which is not usually an issue on embedded devices. If you are curious about the full details, refer to the manual page *shadow(5)*.

In the example, the password for `root` is empty, meaning that `root` can log on without giving a password, which is useful during development, but not for production! You can generate a password hash by using the command `mkpasswd` or by running the `passwd` command on the target and copy and pasting the hash field from `/etc/shadow` on the target into the default shadow file in the staging directory.

The password for `daemon` is `*`, which will not match any logon password, once again ensuring that the `daemon` cannot be used as a regular user account.

Group names are stored in a similar way in `/etc/group`. The format is as follows:

- The name of the group
- The group password, usually an `x` character, indicating that there is no group password
- The GID
- An optional list of users who belong to this group, separated by commas.

Here is an example:

```
root:x:0:  
daemon:x:1:
```

Adding user accounts to the root filesystem

Firstly, you have to add to your staging directory `etc/passwd`, `etc/shadow`, and `etc/group`, as shown in the preceding section. Make sure that the permissions of `shadow` are `0600`.

The login procedure is started by a program called `getty`, which is part of BusyBox. You launch it from `inittab` using the keyword `respawn`, which restarts `getty` when a login shell is terminated, so `inittab` should read like this:

```
::sysinit:/etc/init.d/rcS  
::respawn:/sbin/getty 115200 console
```

Then rebuild the ramdisk and try it out using QEMU or BeagleBone Black as before.

Starting a daemon process

Typically, you would want to run certain background processes at start up. Let's take the log daemon, `syslogd`, as an example. The purpose of `syslogd` is to accumulate log messages from other programs, mostly other daemons. Naturally, BusyBox has an applet for that!

Starting the daemon is as simple as adding a line like this to `etc/inittab`:

```
::respawn:/sbin/syslogd -n
```

`respawn` means that, if the program terminates, it will be automatically restarted; `-n` means that it should run as a foreground process. The log is written to `/var/log/messages`.



You may also want to start `klogd` in the same way: `klogd` sends kernel log messages to `syslogd` so that they can be logged to permanent storage.

As an aside, I should mention that, in the case of a typical embedded Linux system, writing log files to flash memory is not such a good idea as it will wear it out. I will cover the options for logging in *Chapter 7, Creating a Storage Strategy*.

A better way of managing device nodes

Creating device nodes statically with `mknod` is quite hard work and inflexible. There are other ways to create device nodes automatically on demand:

- `devtmpfs`: This is a pseudo filesystem that you mount over `/dev` at boot time. The kernel populates it with device nodes for all the devices that the kernel currently knows about and creates nodes for new devices as they are detected at runtime. The nodes are owned by `root` and have default permissions of `0600`. Some well-known device nodes, such as `/dev/null` and `/dev/random`, override the default to `0666` (see `struct memdev` in `drivers/char/mem.c`).
- `mdev`: This is a BusyBox applet that is used to populate a directory with device nodes and to create new nodes as needed. There is a configuration file, `/etc/mdev.conf`, which contains rules for ownership and the mode of the nodes.
- `udev`: This is now part of `systemd` and is the solution you will find on desktop Linux and some embedded devices. It is very flexible and a good choice for higher end embedded devices.



Although both mdev and udev create the device nodes themselves, it is more usual to let devtmpfs do that job and use mdev/udev as a layer on top to implement the policy for setting ownership and permissions.

An example using devtmpfs

If you have booted up one of the earlier ramdisk examples, trying out devtmpfs is as simple as entering this command:

```
# mount -t devtmpfs devtmpfs /dev
```

You should see that /dev is full of device nodes. For a permanent fix, add this to /etc/init.d/rcS:

```
#!/bin/sh
mount -t proc proc /proc
mount -t sysfs sysfs /sys
mount -t devtmpfs devtmpfs /dev
```

In point of fact, kernel initialization does this automatically unless you have supplied an initramfs ramdisk as we have done! To see the code, look in the `init/do_mounts.c`, function `prepare_namespace()`.

An example using mdev

While mdev is a bit more complex to set up, it does allow you to modify the permissions of device nodes as they are created. Firstly, there is a startup phase, selected by the `-s` option, when mdev scans the /sys directory looking for information about current devices and populates the /dev directory with the corresponding nodes.

If you want to keep track of new devices coming on line and create nodes for them as well, you need to make mdev a hotplug client by writing to /proc/sys/kernel/hotplug. These additions to /etc/init.d/rcS will achieve all of that:

```
#!/bin/sh
mount -t proc proc /proc
mount -t sysfs sysfs /sys
mount -t devtmpfs devtmpfs /dev
echo /sbin/mdev > /proc/sys/kernel/hotplug
mdev -s
```

The default mode is 660 and ownership is `root:root`. You can change that by adding rules in `/etc/mdev.conf`. For example, to give the `null`, `random`, and `urandom` devices their correct modes, you would add this to `/etc/mdev.conf`:

```
null      root:root 666
random    root:root 444
urandom   root:root 444
```

The format is documented in the BusyBox source code in `docs/mdev.txt` and there are more examples in the directory named `examples`.

Are static device nodes so bad after all?

Statically created device nodes do have one advantage: they don't take any time during boot to create, whereas the other methods do. If minimizing boot time is a priority, using statically-created device nodes will save a measurable amount of time.

Configuring the network

Next, let's look at some basic network configurations so that we can communicate with the outside world. I am assuming that there is an Ethernet interface, `eth0`, and that we only need a simple IP v4 configuration.

These examples use the network utilities that are part of BusyBox, and are sufficient for a simple use case, using the old-but-reliable `ifup` and `ifdown` programs. You can read the man pages on both for more details. The main network configuration is stored in `/etc/network/interfaces`. You will need to create these directories in the staging directory:

```
etc/network
etc/network/if-pre-up.d
etc/network/if-up.d
var/run
```

For a static IP address, `etc/network/interfaces` would look like this:

```
auto lo
iface lo inet loopback
auto eth0
iface eth0 inet static
    address 10.0.0.42
    netmask 255.255.255.0
    network 10.0.0.0
```

For a dynamic IP address allocated using DHCP, `/etc/network/interfaces` would look like this:

```
auto lo
iface lo inet loopback
auto eth0
iface eth0 inet dhcp
```

You will also have to configure a DHCP client program. BusyBox has one named `udhcpcd`. It needs a shell script that should go in `/usr/share/udhcpc/default.script`. There is a suitable default in the BusyBox source code in the directory `examples/udhcp/simple.script`.

Network components for glibc

glibc uses a mechanism known as the **name service switch (NSS)** to control the way that names are resolved to numbers for networking and users. User names, for example, may be resolved to UIDs via the file `/etc/passwd`; network services such as HTTP can be resolved to the service port number via `/etc/services`, and so on. All this is configured by `/etc/nsswitch.conf`, see the manual page, `nss(5)` for full details. Here is a simple example that will suffice for most embedded Linux implementations:

```
passwd:      files
group:       files
shadow:      files
hosts:       files dns
networks:    files
protocols:   files
services:    files
```

Everything is resolved by the correspondingly named file in `/etc`, except for the host names, which may additionally be resolved by a DNS lookup.

To make this work, you need to populate `/etc` with those files. Networks, protocols, and services are the same across all Linux systems, so they can be copied from `/etc` in your development PC. `/etc/hosts` should, at least contain, the loopback address:

```
127.0.0.1 localhost
```

We will come to the others, `passwd`, `group`, and `shadow`, later.

The last piece of the jigsaw is the libraries that perform the name resolution. They are plugins that are loaded as needed based on the contents of `nsswitch.conf`, meaning that they do not show up as dependencies if you use `readelf` or similar. You will simply have to copy them from the toolchain's `sysroot`:

```
$ cd ~/rootfs  
$ cp -a $TOOLCHAIN_SYSROOT/lib/libnss* lib  
$ cp -a $TOOLCHAIN_SYSROOT/lib/libresolv* lib
```

Creating filesystem images with device tables

The kernel has a utility, `gen_init_cpio`, that creates a `cpio` file based on format instructions set out in a text file, called a `device table`, which allows a non-root user to create device nodes and to allocate arbitrary UID and GID values to any file or directory.

The same concept has been applied to tools that create other filesystem image formats:

- `jffs2:mkfs.jffs2`
- `ubifs:mkfs.ubifs`
- `ext2:genext2fs`

We will look at `jffs2` and `ubifs` in *Chapter 7, Creating a Storage Strategy*, when we look at filesystems for flash memory. The third, `ext2`, is a fairly old format for hard drives.

They each take a device table file with the format `<name> <type> <mode> <uid> <gid> <major> <minor> <start> <inc> <count>` in which the following applies:

- `name`: Filename
- `type`: One of the following:
 - `f`: A regular file
 - `d`: A directory
 - `c`: A character special device file
 - `b`: A block special device file
 - `p`: A FIFO (named pipe)
- `uid`: The UID of the file
- `gid`: The GID of the file

- `major` and `minor`: the device numbers (device nodes only)
- `start`, `inc`, and `count`: (device nodes only) allow you to create a group of device nodes starting from the `minor` number in `start`

You do not have to specify every file, as with `gen_init_cpio`: you just have to point them at a directory – the staging directory – and list the changes and exceptions you need to make in the final filesystem image.

A simple example which populates static device nodes for us is as follows:

```
/dev      d  755  0   0 - - - - - -  
/dev/null  c  666  0   0   1   3   0   0 -  
/dev/console c  600  0   0   5   1   0   0 -  
/dev/tty00   c  600  0   0 252  0   0   0 -
```

Then, use `genext2fs` to generate a filesystem image of 4 MiB (that is 4,096 blocks of the default size, 1,024 bytes):

```
$ genext2fs -b 4096 -d rootfs -D device-table.txt -U rootfs.ext2
```

Now, you can copy the resulting image, `rootfs.ext2`, to an SD card or similar.

Putting the root filesystem onto an SD card

This is an example of mounting a filesystem from a normal block device, such as an SD card. The same principles apply to other filesystem types and we will look at them in more detail in *Chapter 7, Creating a Storage Strategy*.

Assuming that you have a device with an SD card, and that the first partition is used for the boot files, `MLO` and `u-boot.img` – as on a BeagleBone Black. Assume also that you have used `genext2fs` to create a filesystem image. To copy it to the SD card, insert the card and identify the block device it has been assigned: typically `/dev/sd` or `/dev/mmcblk0`. If it is the latter, copy the filesystem image to the second partition:

```
$ sudo dd if=rootfs.ext2 of=/dev/mmcblk0p2
```

Then, slot the SD card into the device, and set the kernel command line to `root=/dev/mmcblk0p2`. The complete boot sequence is as follows:

```
fatload mmc 0:1 0x80200000 zImage  
fatload mmc 0:1 0x80f00000 am335x-boneblack.dtb  
setenv bootargs console=tty00,115200 root=/dev/mmcblk0p2  
bootz 0x80200000 - 0x80f00000
```

Mounting the root filesystem using NFS

If your device has a network interface, it is best to mount the root filesystem over the network during development. It gives you access to almost unlimited storage so you can add in debug tools and executables with large symbol tables. As an added bonus, updates made to the root filesystem hosted on the development machine are made available on the target immediately. You also have a copy of log files.

For this to work, your kernel has to be configured with `CONFIG_ROOT_NFS`. Then, you can configure Linux to do the mount at boot time by adding the following to the kernel command line:

```
root=/dev/nfs
```

Give the details of the NFS export as follows:

```
nfsroot=<host-ip>:<root-dir>
```

Configure the network interface that connects to the NFS server so that it is available at boot time, before the `init` program runs by using this command:

```
ip=<target-ip>
```

There is more information about NFS root mounts in the kernel source in `Documentation/filesystems/nfs/nfsroot.txt`.

You also need to install and configure an NFS server on your host which, for Ubuntu, you can do with this command:

```
$ sudo apt-get install nfs-kernel-server
```

The NFS server needs to be told which directories are being exported to the network, which is controlled by `/etc/exports`. Add a line like this one to that file:

```
/<path to staging> *(rw, sync, no_subtree_check, no_root_squash)
```

Then, restart the server to pick up the change which, for Ubuntu, is:

```
$ sudo /etc/init.d/nfs-kernel-server restart
```

Testing with QEMU

The following script creates a virtual network between the network device `tap0` on the host and `eth0` on the target using a pair of static IPv4 addresses and then launches QEMU with the parameters to use `tap0` as the emulated interface. You will need to change the path to the root filesystem to be the full path to your staging directory, and maybe the IP addresses if they conflict with your network configuration:

```
#!/bin/bash

KERNEL=zImage
DTB=vexpress-v2p-ca9.dtb
ROOTDIR=/home/chris/rootfs

HOST_IP=192.168.1.1
TARGET_IP=192.168.1.101
NET_NUMBER=192.168.1.0
NET_MASK=255.255.255.0

sudo tunctl -u $(whoami) -t tap0
sudo ifconfig tap0 ${HOST_IP}
sudo route add -net ${NET_NUMBER} netmask ${NET_MASK} dev tap0
sudo sh -c "echo 1 > /proc/sys/net/ipv4/ip_forward"

QEMU_AUDIO_DRV=none \
qemu-system-arm -m 256M -nographic -M vexpress-a9 -kernel $KERNEL \
-append "console=ttyAMA0 root=/dev/nfs rw" \
nfsroot=${HOST_IP}:${ROOTDIR} ip=${TARGET_IP} -dtb ${DTB} -net \
nic -net tap,ifname=tap0,script=no
```

The script is available as `run-qemu-nfs.sh`.

It should boot up as before, but now using the staging directory directly via the NFS export. Any files that you create in that directory will be immediately visible to the target device and any files created in the device will be visible to the development PC.

Testing with BeagleBone Black

In a similar way, you can enter these commands at the U-Boot prompt of the BeagleBone Black:

```
setenv serverip 192.168.1.1
setenv ipaddr 192.168.1.101
setenv npath [path to staging directory]
setenv bootargs console=ttyO0,115200 root=/dev/nfs rw
nfsroot=${serverip}:${npath} ip=${ipaddr}
```

Then, to boot it, load the kernel and dtb from sdcard, as before:

```
fatload mmc 0:1 0x80200000 zImage
fatload mmc 0:1 0x80f00000 am335x-boneblack.dtb
bootz 0x80200000 - 0x80f00000
```

Problems with file permissions

The files that were already in the staging directory are owned by you and will show up on the target when you run `ls -l` with whatever your UID is, typically 1,000. Any files created by the target device will be owned by root. The whole thing is a mess.

Unfortunately, there is no simple way out. The best advice is to make a copy of the staging directory and change ownership to `root:root` (using `sudo chown -R 0:0 *`) and export this directory as the NFS mount. It reduces the inconvenience of having just one copy of the root filesystem shared between development and target systems.

Using TFTP to load the kernel

When working with real hardware such as the BeagleBone Black, it is best to load the kernel over the network, especially when the root filesystem is mounted via NFS. In this way, you are not using any local storage on the device. It saves time if you don't have to keep re-flashing the memory and means that you can get work done while the flash storage drivers are still being developed (it happens).

U-Boot has supported the **Trivial File Transfer Protocol (TFTP)** for many years. Firstly, you need to install a `tftp` daemon on your development machine. On Ubuntu, you would install the `tftpd-hpa` package, which grants read access to files in the directory `/var/lib/tftpboot` to `tftp` clients like U-Boot.

Assuming that you have copied `zImage` and `am335x-boneblack.dtb` into `/var/lib/tftpboot`, enter these commands into U-Boot to load and boot:

```
setenv serverip 192.168.1.1
setenv ipaddr 192.168.1.101
tftpboot 0x80200000 zImage
tftpboot 0x80f00000 am335x-boneblack.dtb
setenv npath [path to staging]
setenv bootargs console=ttyO0,115200 root=/dev/nfs rw
nfsroot=${serverip}:${npath} ip=${ipaddr}
bootz 0x80200000 - 0x80f00000
```

It is fairly common for the response to `tftpboot` to look like this:

```
setenv ipaddr 192.168.1.101
nova!> setenv serverip 192.168.1.1
nova!> tftpboot 0x80200000 zImage
link up on port 0, speed 100, full duplex
Using cpsw device
TFTP from server 192.168.1.1; our IP address is 192.168.1.101
Filename 'zImage'.
Load address: 0x80200000
Loading: T T T T
```

The row of `T` characters on the last line indicate that there is something wrong and the TFTP requests are timing out. The most common reasons are as follows:

- Incorrect IP address for server.
- TFTP daemon not running on server.
- Firewall on server is blocking the TFTP protocol. Most firewalls do indeed block the TFTP port, 69, by default.

In this case, the `tftp` daemon was not running, so I started it with the following command:

```
$ sudo service tftpd-hpa restart
```

Additional reading

- *Filesystem Hierarchy Standard*, currently at version 3.0 available at <http://refspecs.linuxfoundation.org/fhs.shtml>.
- *ramfs, rootfs and initramfs*, Rob Landley, October 17, 2005, which is part of the Linux source code available at <Documentation/filesystems/ramfs-rootfs-initramfs.txt>.

Summary

One of the strengths of Linux is that it can support a wide range of root filesystems which allow it to be tailored to suit a wide range of needs. We have seen that it is possible to construct a simple root filesystem manually with a small number of components, and that BusyBox is especially useful in this regard. By going through the process one step at a time, it has given us insight into some of the basic workings of Linux systems, including network configuration and user accounts. However, the task rapidly becomes unmanageable as devices get more complex. And, there is the ever-present worry that there may be a security hole in the implementation which we have not noticed. In the next chapter, we will look at using embedded build systems to help us out.

6

Selecting a Build System

The preceding chapters covered the four elements of embedded Linux and showed you, step-by-step, how to build a toolchain, a bootloader, a kernel, and a root filesystem, and then combine them into a basic embedded Linux system. And there are a lot of steps! Now it is time to look at ways to simplify the process by automating it as much as possible. I will look at how embedded build systems can help, and look at two in particular: Buildroot and the Yocto Project. Both are complex and flexible tools which would require an entire book to adequately describe how they work. In this chapter, I only want to show you the general ideas behind build systems. I will show you how to build a simple device image to get an overall feel of the system and then how to make some useful changes, using the Nova board example from the previous chapters.

No more rolling your own embedded Linux

The process of creating a system manually, as described in *Chapter 5, Building a Root Filesystem*, is called the **roll your own** (RYO) process. It has the advantage that you are in complete control of the software and you can tailor it to do anything you like. If you want it to do something truly odd but innovative, or if you want to reduce the memory footprint to the smallest possible, RYO is the way to go. But, in the vast majority of situations, building manually is a waste of time and produces inferior, unmaintainable systems.

They are usually built incrementally over a period of months, often undocumented and seldom recreated from scratch because nobody had a clue where each part came from.

Build systems

The idea of a build system is to automate all the steps I have described up to this point. A build system should be able to build, from upstream source code, some or all of the following:

- The toolchain
- The bootloader
- The kernel
- The root filesystem

Building from an upstream source is important for a number of reasons. It means that you have peace of mind that you can rebuild at any time, without external dependencies. It also means that you have the source code for debugging and that you can meet your license requirements to distribute that to users where necessary.

Therefore to do its job, a build system has to be able to do the following:

- Download a source from upstream, either directly from the source code control system or as an archive, and cache it locally
- Apply patches to enable cross compilation, fix architecture-dependent bugs, apply local configuration policies, and so on
- Build the various components
- Create a staging area and assemble a root filesystem
- Create image files in various formats ready to be loaded onto the target

Other things that are useful are as follows:

- Add your own packages containing, for example, applications or kernel changes
- Select various root filesystem profiles: large or small, with and without graphics or other features
- Create a standalone SDK that you can distribute to other developers so that they don't have to install the complete build system
- Track which open source licenses are used by the various packages you have selected
- Allow you to create updates for in-field updating
- Have a user-friendly user interface

In all cases, they encapsulate the components of a system into packages, some for the host and some for the target. Each package is defined by a set of rules to get the source, build it, and install the results in the correct location. There are dependencies between the packages and a build mechanism to resolve the dependencies and build the set of packages required.

Open source build systems have matured considerably over the last few years. There are many around, including:

- **Buildroot:** An easy-to-use system using GNU make and Kconfig (<http://buildroot.org>)
- **EmbToolkit:** A simple system for generating root filesystems; the only one at the time of writing that supports LLVM/Clang out of the box (<https://www.embtoolkit.org>)
- **OpenEmbedded:** A powerful system which is also a core component of the Yocto Project and others (<http://openembedded.org>)
- **OpenWrt:** A build tool oriented towards building firmware for wireless routers (<https://openwrt.org>)
- **PTXdist:** An open source build system sponsored by Pengutronix (http://www.pengutronix.de/software/ptxdist/index_en.html)
- **Tizen:** A comprehensive system, with emphasis on mobile, media, and in-vehicle devices (<https://www.tizen.org>)
- **The Yocto Project:** This extends the OpenEmbedded core with configuration, layers, tools, and documentation: probably the most popular system (<http://www.yoctoproject.org>)

I will concentrate on two of these: Buildroot and The Yocto Project. They approach the problem in different ways and with different objectives.

Buildroot has the primary aim of building root filesystem images, hence the name, although it can build bootloader and kernel images as well. It is easy to install and configure, and generates target images quickly.

The Yocto Project, on the other hand, is more general in the way it defines the target system and so it can build fairly complex embedded devices. Every component is generated as a package in RPM, .deb or .ipk format (see the following section) and then the packages are combined together to make the filesystem image. Furthermore, you can install a package manager in the filesystem image, which allows you to update packages at runtime. In other words, when you build with the Yocto Project, you are, in effect, creating your own custom Linux distribution.

Package formats and package managers

Mainstream Linux distributions are, in most cases, constructed from collections of binary (precompiled) packages in either RPM or deb format. **RPM** stands for **Red Hat Package Manager** and is used in Red Hat, Suse, Fedora, and other distributions based on them. Debian-derived distributions, including Ubuntu and Mint, use the Debian package manager format, deb. In addition, there is a light-weight format specific to embedded devices known as the **Itsy PacKage** format, or **ipk**, which is based on deb.

The ability to include a package manager on the device is one of the big differentiators between build systems. Once you have a package manager on the target device, you have an easy path to deploy new packages to it and to update existing ones. I will talk about the implications of this in the next chapter.

Buildroot

The Buildroot project website is at <http://buildroot.org>.

Current versions of Buildroot are capable of building a toolchain, a bootloader (U-Boot, Barebox, GRUB2, or Gummiboot), a kernel, and a root filesystem. It uses GNU make as the principal build tool.

There is good online documentation at <http://buildroot.org/docs.html>, including *The Buildroot User Manual*.

Background

Buildroot was one of the first build systems. It began as part of the uClinux and uClibc projects as a way of generating a small root filesystem for testing. It became a separate project in late 2001 and continued to evolve through to 2006, after which it went into a rather dormant phase. However, since 2009, when Peter Korsgaard took over stewardship, it has been developing rapidly, adding support for glibc-based toolchains and the ability to build a bootloader and a kernel.

Buildroot is also the foundation of another popular build system, OpenWrt (<http://wiki.openwrt.org>) which forked from Buildroot around 2004. The primary focus of OpenWrt is to produce software for wireless routers and so the package mix is oriented towards networking infrastructure. It also has a runtime package manager using the .ipk format so that a device can be updated or upgraded without a complete re-flash of the image.

Stable releases and support

The Buildroot developers produce stable releases four times a year, in February, May, August, and November. They are marked by git tags of the form <year>.02, <year>.05, <year>.08, and <year>.11. Typically, when you start your project, you will be using the latest stable release. However, the stable releases are seldom updated after release. To get security fixes and other bug fixes you will have to either continually update to the next stable release as they become available or backport the fixes into your version.

Installing

As usual, you can install Buildroot either by cloning the repository or downloading an archive. Here is an example of obtaining version 2015.08.1, which was the latest stable version at the time of writing:

```
$ git clone git://git.buildroot.net/buildroot
$ cd buildroot
$ git checkout 2015.08.1
```

The equivalent TAR archive is available from <http://buildroot.org/downloads>.

Next, you should read the section titled *System Requirement* from *The Buildroot User Manual*, available at <http://buildroot.org/downloads/manual/manual.html> and make sure that you have installed all the packages listed there.

Configuring

Buildroot uses the Kconfig and Kbuild mechanisms as the kernel, which I described in the section *Understanding kernel configuration* in *Chapter 4, Porting and Configuring the Kernel*. You can configure it from scratch directly using make menuconfig (or xconfig or gconfig), or you can choose one of the 90 or so configurations for various development boards and the QEMU emulator which you can find stored in the directory configs/. Typing make help lists all the targets including the default configurations.

Let's begin by building a default configuration that you can run on the ARM QEMU emulator:

```
$ cd buildroot
$ make qemu_arm_versatile_defconfig
$ make
```



Note that you do not tell make how many parallel jobs to run with a -j option: Buildroot will make optimum use of your CPUs all by itself. If you want to limit the number of jobs, you can run make menuconfig and look under **Build** options.

The build will take half an hour to an hour, depending on the capabilities of your host system and the speed of your link to the Internet. When it is complete, you will find that two new directories have been created:

- dl/: This contains archives of the upstream projects that Buildroot has built
- output/: This contains all the intermediate and final compiled resources

You will see the following in output/:

- build/: This is the build directory for each component.
- host/: This contains various tools required by Buildroot that run on the host, including the executables of the toolchain (in output/host/usr/bin).
- images/: This is the most important of all and contains the results of the build. Depending on what you selected when configuring, you will find a bootloader, a kernel, and one or more root filesystem images.
- staging/: This is a symbolic link to the sysroot of the toolchain. The name of the link is a little confusing because it does not point to a staging area as I defined it in *Chapter 5, Building a Root Filesystem*.
- target/: This is the staging area for the root directory. Note that you cannot use this as a root filesystem, as it stands, because the file ownership and permissions are not set correctly. Buildroot uses a device table, as described in the previous chapter, to set ownership and permissions when the filesystem image is created.

Running

Some of the sample configurations have a corresponding entry in the directory boards/, which contains custom configuration files and information about installing the results on the target. In the case of the system you have just built, the relevant file is board/qemu/arm-vexpress/readme.txt, which tells you how to start QEMU with this target.

Assuming that you have already installed `qemu-system-arm` as described in *Chapter 1, Starting Out*, you can run it using this command:

```
$ qemu-system-arm -M vexpress-a9 -m 256 \
-kernel output/images/zImage \
-dtb output/images/vexpress-v2p-ca9.dtb \
-drive file=output/images/rootfs.ext2,if=sd \
-append "console=ttyAMA0,115200 root=/dev/mmcblk0" \
-serial stdio -net nic,model=lan9118 -net user
```

You should see the kernel boot messages appear in the same terminal window where you started QEMU, followed by a login prompt:

```
Booting Linux on physical CPU 0x0
Initializing cgroup subsys cpuset

Linux version 4.1.0 (chris@builder) (gcc version 4.9.3 (Buildroot
2015.08) ) #1 SMP Fri Oct 30 13:55:50 GMT 2015

CPU: ARMv7 Processor [410fc090] revision 0 (ARMv7), cr=10c5387d

CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction
cache
Machine model: V2P-CA9
[...]
VFS: Mounted root (ext2 filesystem) readonly on device 179:0.
devtmpfs: mounted
Freeing unused kernel memory: 264K (8061e000 - 80660000)
random: nonblocking pool is initialized
Starting logging: OK
Starting mdev...
Initializing random number generator... done.
Starting network...

Welcome to Buildroot
buildroot login:

Log in as root, no password.
```

You will see that QEMU launches a black window in addition to the one with the kernel boot messages. It is there to display the graphics frame buffer of the target. In this case, the target never writes to the framebuffer, which is why it appears black. To close QEMU, either type poweroff at the root prompt or just close the framebuffer window. This works with QEMU 2.0 (default on Ubuntu 14.04), but fails with earlier versions including QEMU 1.0.50 (default on Ubuntu 12.04) because of problems with the SCSI emulation.

Creating a custom BSP

Next, let's use Buildroot to create a BSP for our Nova board, using the same versions of U-Boot and Linux from earlier chapters. The recommended places to store your changes are:

- `board/<organization>/<device>`: contains any patches, binary blobs, extra build steps, configuration files for Linux, U-Boot, and other components
- `configs/<device>_defconfig`: contains the default configuration for the board
- `packages/<organization>/<package_name>`: is the place to put any additional packages for this board

We can use the BeagleBone configuration file as a base, since Nova is a close cousin:

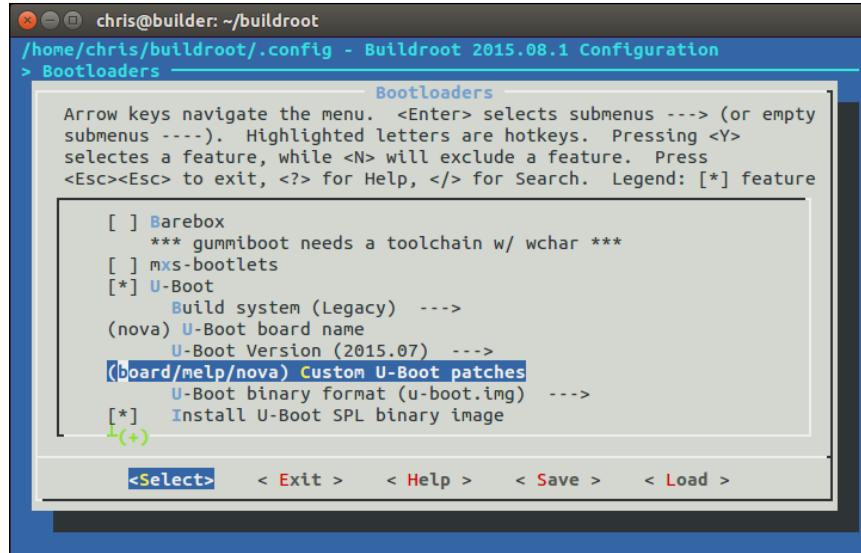
```
$ make clean # Always do a clean when changing targets  
$ make beaglebone_defconfig
```

Now the `.config` file is set for BeagleBone. Next, create a directory for the board configuration:

```
$ mkdir -p board/melp/nova
```

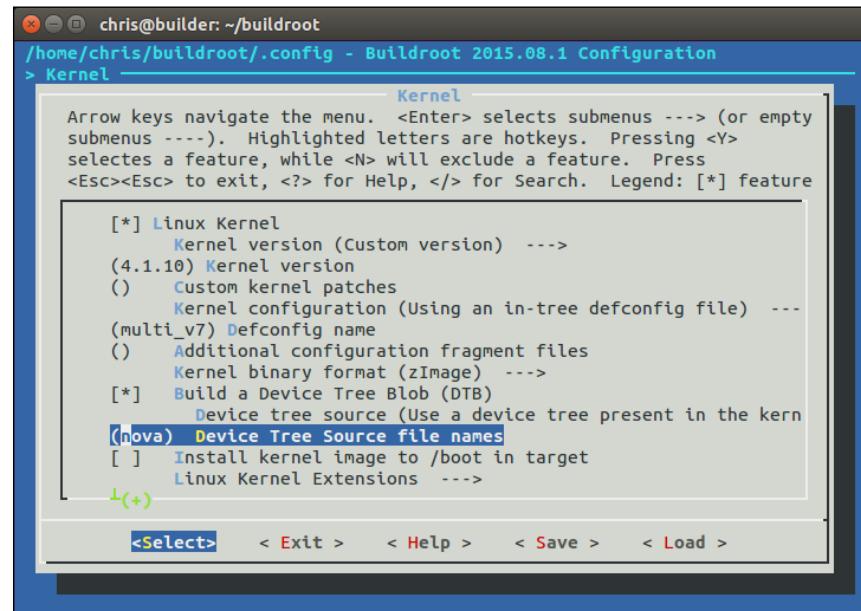
U-Boot

In *Chapter 3, All About Bootloaders*, we created a custom bootloader for Nova, based on the 2015.07 of U-Boot version and created a patch file for it. We can configure Buildroot to select the same version, and apply our patch. Begin by copying the patch file into `board/melp/nova`, and then use `make menuconfig` to set the U-Boot version to 2015.07, the patch directory to `board/melp/nova` and the board name to `nova`, as shown in this screenshot:



Linux

In *Chapter 4, Porting and Configuring the Kernel*, we based the kernel on Linux 4.1.10 and supplied a new device tree, named `nova.dts`. Copy the device tree to `board/melp/nova` and change the Buildroot kernel configuration to use this version and the `nova` device tree as shown in this screenshot:



Build

Now you can build the system for the Nova board just by typing `make`, which produces these files in the directory `output/images`:

```
MLO  nova.dtb  rootfs.ext2  u-boot.img  uEnv.txt  zImage
```

The last step is to save a copy of the configuration so that you and others can use it again:

```
$ make savedefconfig BR2_DEFCONFIG=configs/nova_defconfig
```

Now, you have a Buildroot configuration for the Nova board.

Adding your own code

Suppose that there are some programs that you have developed that you want to include in the build. You have two options: firstly to build them separately, using their own build systems, and then roll the binary into the final build as an overlay. Secondly you could create a Buildroot package that can be selected from the menu and built like any other.

Overlay

An overlay is simply a directory structure that is copied over the top of the Buildroot root filesystem at a late stage in the build process. It can contain executables, libraries and anything else you may want to include. Note that any compiled code must be compatible with the libraries deployed at runtime, which means that it must be compiled with the same toolchain that Buildroot uses. Using the Buildroot toolchain is quite easy: just add it to the path:

```
$ PATH=<path_to_buildroot>/output/host/usr/bin:$PATH
```

The prefix for the tools is `<ARCH>-linux-`.

The overlay directory is set by `BR2_ROOTFS_OVERLAY`, which contains a list of directories separated by spaces, which you should overlay on the Buildroot root filesystem. It can be configured in `menuconfig` with the option **System configuration | Root filesystem overlay directories**.

For example, if you add a `helloworld` program to the `bin` directory, and a script to start it at boot time, you would create an overlay directory with the following contents:

```
board/melp/nova/overlay/
├── bin
│   └── helloworld
└── etc
    └── init.d
        └── s99helloworld
```

Then you would add `board/melp/nova/overlay` to the overlay options.

The layout of the root filesystem is controlled by the `system/skeleton` directory, and the permissions are set in `device_table_dev.txt` and `device_table.txt`.

Adding a package

Buildroot packages are stored in the package directory, over 1,000 of them, each in its own subdirectory. A package consists of at least two files: `Config.in`, containing the snippet of `Kconfig` code required to make the package visible in the **configuration** menu, and a `makefile` named `<package_name>.mk`. Note that the package does not contain the code, just the instructions to get the code by downloading a tarball, doing a `git pull`, and so on.

The `makefile` is written in a format expected by Buildroot and contains directives that allow Buildroot to download, configure, compile, and install the program. Writing a new package `makefile` is a complex operation which is covered in detail in the *Buildroot User Manual*. Here is an example which shows you how to create a package for a simple program stored locally, such as our `helloworld` program.

Begin by creating the subdirectory `package/helloworld` with a configuration file, `Config.in`, that looks like this:

```
config BR2_PACKAGE_HELLOWORLD
bool "helloworld"
help
A friendly program that prints Hello World! every 10s
```

The first line must be of the format `BR2_PACKAGE_<uppercase package name>`. That is followed by a Boolean and the package name as it will appear in the **configuration** menu and which will allow a user to select this package. The *Help* section is optional (but hopefully useful).

Next, link the new package into the **Target Packages** menu by editing `package/Config.in` and sourcing the configuration file as mentioned in the preceding section. You could append this to an existing sub-menu but, in this case, it seems neater to create a new sub-menu which only contains our package:

```
menu "My programs"
    source "package/helloworld/Config.in"
endmenu
```

Then, create a makefile, `package/helloworld/helloworld.mk`, to supply the data needed by Buildroot:

```
HELLOWORLD_VERSION:= 1.0.0
HELLOWORLD_SITE:= /home/chris/MELP/helloworld/
HELLOWORLD_SITE_METHOD:=local
HELLOWORLD_INSTALL_TARGET:=YES

define HELLOWORLD_BUILD_CMDS
    $(MAKE) CC=$(TARGET_CC) LD=$(TARGET_LD) -C $(@D) all
endef

define HELLOWORLD_INSTALL_TARGET_CMDS
    $(INSTALL) -D -m 0755 $(@D)/helloworld $(TARGET_DIR)/bin
endef

$(eval $(generic-package))
```

The location of the code is hard-coded to a local path name. In a more realistic case, you would get the code from a source code system or from a central server of some kind: there are details of how to do this in the *Buildroot User Guide* and plenty of examples in other packages.

License compliance

Buildroot is based on open source software, as are the packages it compiles. At some point during the project, you should check the licenses, which you can do by running:

```
$ make legal-info
```

The information is gathered into `output/legal-info`. There are summaries of the licenses used to compile the host tools in `host-manifest.csv` and, on the target, in `manifest.csv`. There is more information in the `README` file and in the *Buildroot User Manual*.

The Yocto Project

The Yocto Project is a more complex beast than Buildroot. Not only can it build toolchains, bootloaders, kernels, and root filesystems, as Buildroot can, but it can generate an entire Linux distribution for you, with binary packages that can be installed at runtime.

The Yocto Project is primarily a group of recipes, similar to Buildroot packages but written using a combination of Python and shell script, and a task scheduler called BitBake that produces whatever you have configured, from the recipes.

There is plenty of online documentation at <https://www.yoctoproject.org/>.

Background

The structure of the Yocto Project makes more sense if you look at the background first. Its roots are in OpenEmbedded, <http://openembedded.org/> which, in turn, grew out of a number of projects to port Linux to various hand-held computers, including the Sharp Zaurus and Compaq iPaq. OpenEmbedded came to life in 2003 as the build system for those hand-held computers but quickly expanded to encompass other embedded boards. It was developed and continues to be developed by an enthusiastic community of programmers.

The OpenEmbedded project set out to create a set of binary packages using the compact `.ipk` format, which could then be combined in various ways to create a target system and be installed on the target at runtime. It did this by creating recipes for each piece of software and using BitBake as the task scheduler. It was, and is, very flexible. By supplying the right metadata, you can create an entire Linux distribution to your own specification. One that is fairly well known is *The Ångström Distribution*, <http://www.angstrom-distribution.org>, but there are many others.

At some time in 2005 Richard Purdie, then a developer at OpenedHand, created a fork of OpenEmbedded which had a more conservative choice of packages and created releases that were stable over a period of time. He named it Poky, after the Japanese snack (if you are worried about these things, Poky is pronounced to rhyme with hockey). Although Poky was a fork, OpenEmbedded and Poky continued to run alongside each other, sharing updates and keeping the architectures more or less in step. Intel brought out OpenedHand in 2008 and they transferred Poky Linux to the Linux Foundation in 2010 when they formed the Yocto Project.

Since 2010, the common components of OpenEmbedded and Poky have been combined into a separate project known as OpenEmbedded core, or just oe-core.

Therefore, the Yocto Project collects together several components, the most important of which are the following:

- **Poky**: The reference distribution
- **oe-core**: The core metadata, which is shared with OpenEmbedded
- **BitBake**: The task scheduler, which is shared with OpenEmbedded and other projects
- **Documentation**: User manuals and developer's guides for each component
- **Hob**: A graphical user interface to OpenEmbedded and BitBake
- **Toaster**: A web-based interface to OpenEmbedded and BitBake
- **ADT Eclipse**: A plug-in for Eclipse that makes it easier to build projects using the Yocto Project SDK

Strictly speaking, the Yocto Project is an umbrella for these sub-projects. It uses OpenEmbedded as its build system, and Poky as its default configuration and reference environment. However, people often use the term "the Yocto Project" to refer to the build system alone. I feel that it is too late for me to turn this tide, so for brevity I will do the same. I apologise in advance to the developers of OpenEmbedded.

The Yocto Project provides a stable base which can be used as it is or which can be extended using meta layers, which I will discuss later in this chapter. Many SoC vendors provide board support packages for their devices in this way. Meta layers can also be used to create extended, or just different, build systems. Some are open source, such as the Angstrom Project, others are commercial, such as MontaVista Carrier Grade Edition, Mentor Embedded Linux, and Wind River Linux. The Yocto Project has a branding and compatibility testing scheme to ensure that there is interoperability between components. You will see statements like *Yocto Project Compatible 1.7* on various web pages.

Consequently, you should think of the Yocto Project as the foundation of a whole sector of embedded Linux, as well as being a complete build system in its own right. You may be wondering about the name, *yocto*. A *yocto* is the SI prefix for 10^{-24} , in the same way that *micro* is 10^{-6} . Why name the project *yocto*? It was partly to indicate that it could build very small Linux systems (although, to be fair, so can other build systems), but also, perhaps, to steal a march on the Ångström distribution which is based on OpenEmbedded. An Ångström is 10^{-10} . That's huge, compared to a *yocto*!

Stable releases and support

Usually, there is a release of the Yocto Project every six months, in April and October. They are principally known by the code name, but it is useful to know the version numbers of the Yocto Project and Poky as well. Here is a table of the four most recent releases at the time of writing:

Code name	Release date	Yocto version	Poky version
Fido	April 2015	1.8	13
Dizzy	October 2014	1.7	12
Daisy	April 2014	1.6	11
Dora	October 2013	1.5	10

The stable releases are supported with security and critical bug fixes for the current release cycle and the next cycle, that is approximately twelve months after release. No toolchain or kernel version changes are allowed for these updates. As with Buildroot, if you want continued support, you can update to the next stable release or you can backport changes to your version. You also also have the option of commercial support for periods of several years with the Yocto Project from operating system vendors such as Mentor Graphics, Wind River, and many others.

Installing the Yocto Project

To get a copy of the Yocto Project, you can either clone the repository, choosing the code name as the branch which is `fido` in this case:

```
$ git clone -b fido git://git.yoctoproject.org/poky.git
```

You can also download the archive from <http://downloads.yoctoproject.org/releases/yocto/yocto-1.8/poky-fido-13.0.0.tar.bz2>.

In the first case, you will find everything in the `poky` directory, in the second case, `poky-fido-13.0.0/`.

In addition, you should read the section titled *System Requirements* from the *Yocto Project Reference Manual* (<http://www.yoctoproject.org/docs/current/ref-manual/ref-manual.html#detailed-supported-distros>) and, in particular, you should make sure that the packages listed there are installed on your host computer.

Configuring

As with Buildroot, let's begin with a build for the ARM QEMU emulator. Begin by sourcing a script to set up the environment:

```
$ cd poky  
$ source oe-init-build-env
```

That creates a working directory for you named `build` and makes it the current directory. All of the configuration, intermediate, and deployable files will be put in this directory. You must source this script each time you want to work on this project.

You can choose a different working directory by adding it as a parameter to `oe-init-build-env`, for example:

```
$ source oe-init-build-env build-qemuarm
```

That will put you into the `build-qemuarm` directory. You can then have several projects on the go at the same time: you choose which one you want to work with through the parameter to `oe-init-build-env`.

Initially, the `build` directory contains only one subdirectory named `conf`, which contains the configuration files for this project:

- `local.conf`: Contains a specification of the device you are going to build and the build environment.
- `bblayers.conf`: Contains a list of the directories that contain the layers you are going to use. There will be more on layers later on.
- `templateconf.cfg`: Contains the name of a directory which contains various `conf` files. By default, it points to `meta-yocto/conf`.

For now, we just need to set the `MACHINE` variable in `local.conf` to `qemuarm` by removing the comment character at the start of this line:

```
MACHINE ?= "qemuarm"
```

Building

To actually perform the build, you need to run `bitbake`, telling it which root filesystem image you want to create. Some common images are as follows:

- **core-image-minimal**: A small console-based system which is useful for tests and as the basis for custom images.
- **core-image-minimal-initramfs**: This is similar to core-image-minimal, but built as a ramdisk.
- **core-image-x11**: A basic image with support for graphics through an X11 server and the xterminal terminal app.
- **core-image-sato**: A full graphical system based on Sato, which is a mobile graphical environment built on X11, and GNOME. The image includes several apps including a terminal, an editor, and a file manager.

By giving BitBake the final target, it will work backwards and build all the dependencies first, beginning with the toolchain. For now, we just want to create a minimal image to see whether or not it works:

```
$ bitbake core-image-minimal
```

The build is likely to take some time, maybe more than an hour. When it is complete, you will find several new directories in the build directory including `build/downloads`, which contains all the source downloaded for the build, and `build/tmp` which contains most of the build artifacts. You should see the following in `tmp`:

- `work`: Contains the build directory and the staging area for all components, including the root filesystem
- `deploy`: Contains the final binaries to be deployed on the target:
 - `deploy/images/ [machine name]`: Contains the bootloader, the kernel, and the root filesystem images ready to be run on the target
 - `deploy/rpm`: Contains the RPM packages that went to make up the images
 - `deploy/licenses`: Contains the license files extracted from each package

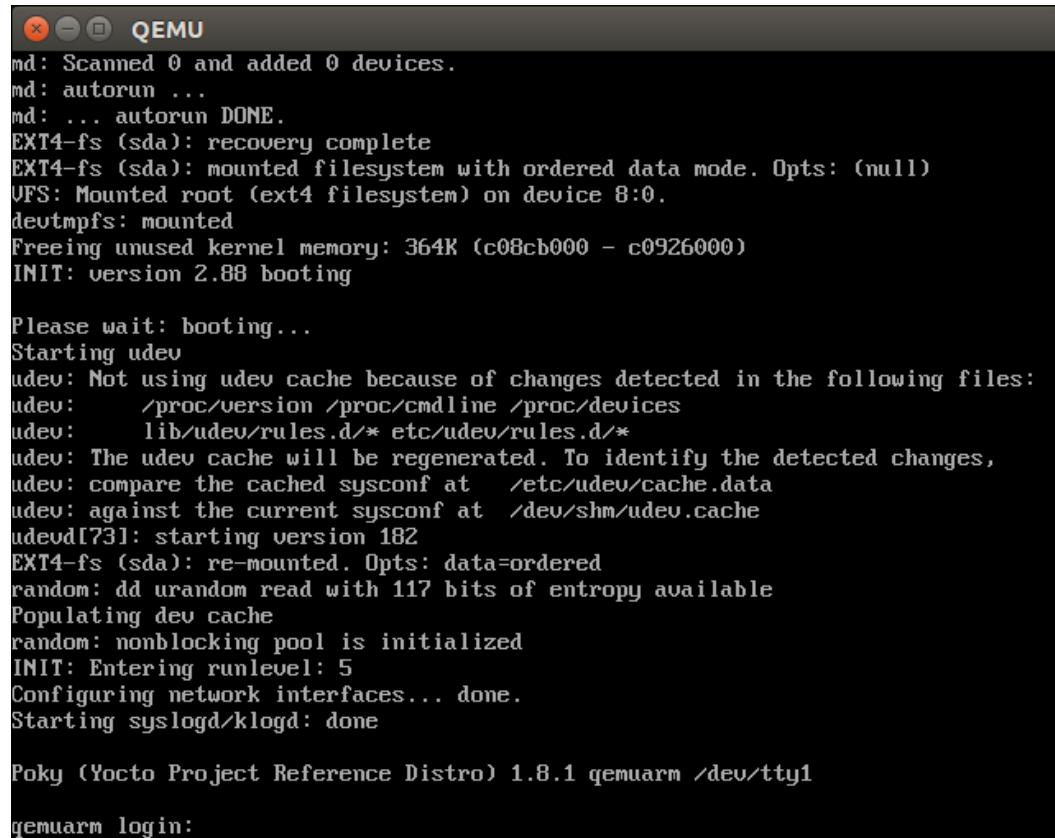
Running

When you build a QEMU target, an internal version of QEMU is generated, which removes the need to install the QEMU package for your distribution and thus avoids version dependencies. There is a wrapper script named `runqemu` for this internal QEMU.

To run the QEMU emulation, make sure that you have sourced `oe-init-build-env` and then just type:

```
$ runqemu qemuarm
```

In this case, QEMU has been configured with a graphic console so that the boot messages and login prompt appear in the black framebuffer screen:



```
QEMU
md: Scanned 0 and added 0 devices.
md: autorun ...
md: ... autorun DONE.
EXT4-fs (sda): recovery complete
EXT4-fs (sda): mounted filesystem with ordered data mode. Opts: (null)
VFS: Mounted root (ext4 filesystem) on device 8:0.
devtmpfs: mounted
Freeing unused kernel memory: 364K (c08cb000 - c0926000)
INIT: version 2.88 booting

Please wait: booting...
Starting udev
udev: Not using udev cache because of changes detected in the following files:
udev: /proc/version /proc/cmdline /proc/devices
udev: lib/udev/rules.d/* etc/udev/rules.d/*
udev: The udev cache will be regenerated. To identify the detected changes,
udev: compare the cached sysconf at /etc/udev/cache.data
udev: against the current sysconf at /dev/shm/udev.cache
udevd[73]: starting version 182
EXT4-fs (sda): re-mounted. Opts: data=ordered
random: dd urandom read with 117 bits of entropy available
Populating dev cache
random: nonblocking pool is initialized
INIT: Entering runlevel: 5
Configuring network interfaces... done.
Starting syslogd/klogd: done

Poky (Yocto Project Reference Distro) 1.8.1 qemuarm /dev/tty1

qemuarm login:
```

You can log on as root, without a password. You can close down QEMU by closing the framebuffer window. You can launch QEMU without the graphic window by adding `nographic` to the command line:

```
$ runqemu qemuarm nographic
```

In this case, you close QEMU using the key sequence *Ctrl + A + X*.

The `runqemu` script has many other options, type `runqemu help` for more information.

Layers

The metadata for the Yocto Project is structured into layers, by convention, each with a name beginning with `meta`. The core layers of the Yocto Project are as follows:

- **meta**: This is the OpenEmbedded core
- **meta-yocto**: Metadata specific to the Yocto Project, including the Poky distribution
- **meta-yocto-bsp**: Contains the board support packages for the reference machines that the Yocto Project supports

The list of layers in which BitBake searches for recipes is stored in `<your build directory>/conf/bblayers.conf` and, by default, includes all three layers mentioned in the preceding list.

By structuring the recipes and other configuration data in this way, it is very easy to extend the Yocto Project by adding new layers. Additional layers are available from SoC manufacturers, the Yocto Project itself, and a wide range of people wishing to add value to the Yocto Project and OpenEmbedded. There is a useful list of layers at <http://layers.openembedded.org>. Here are some examples:

- **meta-angstrom**: The Ångström distribution
- **meta-qt5**: Qt5 libraries and utilities
- **meta-fsl-arm**: BSPs for Freescale ARM-based SoCs
- **meta-fsl-ppc**: BSPs for Freescale PowerPC-based SoCs
- **meta-intel**: BSPs for Intel CPUs and SoCs
- **meta-ti**: BSPs for TI ARM-based SoCs

Adding a layer is as simple as copying the `meta` directory into a suitable location, usually alongside the default `meta` layers, and adding it to `bblayers.conf`. Just make sure it is compatible with the version of the Yocto Project you are using.

To illustrate the way layers work, let's create a layer for our Nova board which we can use for the remainder of the chapter as we add features. Each meta layer has to have at least one configuration file, `conf/layer.conf`, and should also have a `README` file and a license. There is a handy helper script that does the basics for us:

```
$ cd poky  
$ scripts/yocto-layer create nova
```

The script asks for a priority, and if you want to create sample recipes. In the example here, I just accepted the defaults:

```
Please enter the layer priority you'd like to use for the layer:  
[default: 6]  
Would you like to have an example recipe created? (y/n) [default: n]  
Would you like to have an example bbappend file created? (y/n)  
[default: n]  
New layer created in meta-nova.  
Don't forget to add it to your BBLAYERS (for details see meta-nova\README).
```

That will create a layer named `meta-nova` with a `conf/layer.conf`, an outline `README` and a MIT license in `COPYING.MIT`. The `layer.conf` file looks like this:

```
# We have a conf and classes directory, add to BBPATH  
BBPATH .= ":${LAYERDIR}"  
  
# We have recipes-* directories, add to BBFILES  
BBFILES += "${LAYERDIR}/recipes-*/*/*.bb \  
${LAYERDIR}/recipes-*/*/*.bbappend"  
  
BBFILE_COLLECTIONS += "nova"  
BBFILE_PATTERN_nova = "^${LAYERDIR} /"  
BBFILE_PRIORITY_nova = "6"
```

It adds itself to `BBPATH` and the recipes it contains to `BBFILES`. From looking at the code, you can see that the recipes are found in the directories with names beginning `recipes-` and have file names ending in `.bb` (for normal BitBake recipes), or `.bbappend` (for recipes that extend existing normal recipes by adding and overriding instructions). This layer has the name `nova` which is added to the list of layers in `BBFILE_COLLECTIONS` and it has a priority of 6. The layer priority is used if the same recipe appears in several layers: the one in the layer with the highest priority wins.

Since you are about to build a new configuration, it is best to begin by creating a new build directory named `build-nova`:

```
$ cd ~/poky  
$ . oe-init-build-env build-nova
```

Now you need to add this layer to your build configuration, `conf/bblayers.conf`:

```
LCONF_VERSION = "6"  
  
BBPATH = "${TOPDIR}"  
BBFILES ?= ""  
  
BBLAYERS ?= " \  
/home/chris/poky/meta \  
/home/chris/poky/meta-yocto \  
/home/chris/poky/meta-yocto-bsp \  
/home/chris/poky/meta-nova \  
"  
BBLAYERS_NON_REMOVABLE ?= " \  
/home/chris/poky/meta \  
/home/chris/poky/meta-yocto \  
"
```

You can confirm that it is set up correctly by using another helper script:

```
$ bitbake-layers show-layers  
layer          path          priority  
=====
```

meta	/home/chris/poky/meta	5
meta-yocto	/home/chris/poky/meta-yocto	5
meta-yocto-bsp	/home/chris/poky/meta-yocto-bsp	5
meta-nova	/home/chris/poky/meta-nova	6

There you can see the new layer. It has priority 6 which means that we could override recipes in the other layers, which have a lower priority.

At this point it would be a good idea to run a build, using this empty layer. The final target will be the Nova board but, for now, build for a BeagleBone Black by removing the comment before `MACHINE ?= "beaglebone"` in `conf/local.conf`. Then, build a small image using `bitbake core-image-minimal` as before.

As well as recipes, layers may contain BitBake classes, configuration files for machines, distributions, and more. I will look at recipes next and show you how to create a customized image and how to create a package.

BitBake and recipes

BitBake processes metadata of several different types, which include the following:

- **recipes**: Files ending in `.bb`. These contain information about building a unit of software, including how to get a copy of the source code, the dependencies on other components, and how to build and install it.
- **append**: Files ending in `.bbappend`. These allow some details of a recipe to be overridden or extended. A `.bbappend` file simply appends its instructions to the end of a recipe (`.bb`) file of the same root name.
- **include**: Files ending in `.inc`. These contain information that is common to several recipes, allowing information to be shared among them. The files may be included using the `include` or `require` keywords. The difference is that `require` produces an error if the file does not exist, whereas `include` does not.
- **classes**: Files ending in `.bbclass`. These contain common build information, for example how to build a kernel or how to build an `autotools` project. The classes are inherited and extended in recipes and other classes using the `inherit` key word. The class `classes/base.bbclass` is implicitly inherited in every recipe.
- **configuration**: Files ending in `.conf`. They define various configuration variables that govern the project's build process.

A recipe is a collection of tasks written in a combination of Python and shell code. The tasks have names like `do_fetch`, `do_unpack`, `do_patch`, `do_configure`, `do_compile`, `do_install`, and so on. You use BitBake to execute these tasks.

The default task is `do_build`, so that you are running the build task for that recipe. You can list the tasks available in a recipe by running `bitbake core-image-minimal` like this:

```
$ bitbake -c listtasks core-image-minimal
```

The `-c` option allows you to specify the task, missing off the `do_` part. A common use is `-c fetch` to get the code needed by a recipe:

```
$ bitbake -c fetch busybox
```

You can also use `fetchall` to get the code for the target and all the dependencies:

```
$ bitbake -c fetchall core-image-minimal
```

The recipe files are usually named <package-name>.version.bb. They may have dependencies on other recipes, which would allow BitBake to work out all the subtasks that need to be executed to complete the top level job. Unfortunately, I don't have the space in this book to describe the dependency mechanism, but you will find a full description in the Yocto Project documentation.

As an example, to create a recipe for our `helloworld` program in `meta-nova`, you would create a directory structure like this:

```
meta-nova/recipes-local/helloworld
├── files
│   └── helloworld.c
└── helloworld_1.0.bb
```

The recipe is `helloworld_1.0.bb` and the source is local to the recipe directory in the subdirectory `files`. The recipe contains these instructions:

```
DESCRIPTION = "A friendly program that prints Hello World!"
PRIORITY = "optional"
SECTION = "examples"

LICENSE = "GPLv2"
LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/GPL-
2.0;md5=801f80980d171dd6425610833a22dbe6"

SRC_URI = "file://helloworld.c"
S = "${WORKDIR}"

do_compile() {
    ${CC} ${CFLAGS} -o helloworld helloworld.c
}

do_install() {
    install -d ${D}${bindir}
    install -m 0755 helloworld ${D}${bindir}
}
```

The location of the source code is set by `SRC_URI`: in this case it will search directories, `files`, `helloworld`, and `helloworld-1.0` in the recipe directory. The only tasks that need to be defined are `do_compile` and `do_install`, which compile the one source file simply and install it into the target root filesystem: `${D}` expands to the staging area of the target device and `${bindir}` to the default binary directory, `/usr/bin`.

Every recipe has a license, defined by `LICENSE`, which is set to `GPLv2` here. The file containing the text of the license and a checksum is defined by `LIC_FILES_CHKSUM`. BitBake will terminate the build if the checksum does not match, indicating that the license has changed in some way. The license file may be part of the package or it may point to one of the standard license texts in `meta/files/common-licenses`, as is the case here.

By default, commercial licenses are disallowed, but it is easy to enable them. You need to specify the license in the recipe, as shown here:

```
LICENSE_FLAGS = "commercial"
```

Then, in your `conf/local.conf`, you would explicitly allow this license, like so:

```
LICENSE_FLAGS_WHITELIST = "commercial"
```

To make sure that it compiles correctly, you can ask BitBake to build it, like so:

```
$ bitbake helloworld
```

If all goes well, you should see that it has created a working directory for it in `tmp/work/cortexa8hf-vfp-neon-poky-linux-gnueabi/helloworld/`.

You should also see there is an RPM package for it in `tmp/deploy/rpm/cortexa8hf_vfp_neon/helloworld-1.0-r0.cortexa8hf_vfp_neon.rpm`.

It is not part of the target image yet, though. The list of packages to be installed is held in a variable named `IMAGE_INSTALL`. You can append to the end of that list by adding this line to your `conf/local.conf`:

```
IMAGE_INSTALL_append = " helloworld"
```

Note that there has to be a space between the first double quote and the first package name. Now, the package will be added to any image that you `bitbake`:

```
$ bitbake core-image-minimal
```

If you look in `tmp/deploy/images/beaglebone/core-image-minimal-beaglebone.tar.bz2` you will see that `/usr/bin/helloworld` has indeed been installed.

Customizing images via local.conf

You may often want to add a package to an image during development or tweak it in other ways. As shown previously, you can simply append to the list of packages to be installed by adding a statement like this:

```
IMAGE_INSTALL_append = " strace helloworld"
```

It should be no surprise that you can also do the opposite: you can remove a package using this syntax:

```
IMAGE_INSTALL_remove = "someapp"
```

You can make more sweeping changes via `EXTRA_IMAGE_FEATURES`. There are too many to list here, I recommend you look at the *Image Features* section of the *Yocto Project Reference Manual* and the code in `meta/classes/core-image.bbclass`. Here is a short list which should give you an idea of the features you can enable:

- `dbg-pkgs`: installs debug symbol packages for all the packages installed in the image.
- `debug-tweaks`: allows root logins without passwords and other changes that make development easier.
- `package-management`: installs package management tools and preserves the package manager database.
- `read-only-rootfs`: makes the root filesystem read-only. We will cover this in more detail in *Chapter 7, Creating a Storage Strategy*.
- `x11`: installs the X server.
- `x11-base`: installs the X server with a minimal environment.
- `x11-sato`: installs the OpenedHand Sato environment.

Writing an image recipe

The problem with making changes to `local.conf` is that they are, well, local. If you want to create an image that is to be shared with other developers, or to be loaded onto a production system, then you should put the changes into an image recipe.

An image recipe contains instructions about how to create the image files for a target, including the bootloader, the kernel, and the root filesystem images. You can get a list of the images that are available by using this command:

```
$ ls meta*/recipes*/images/*.bb
```

The recipe for `core-image-minimal` is in `meta/recipes-core/images/core-image-minimal.bb`.

A simple approach is to take an existing image recipe and modify it using statements similar to those you used in `local.conf`.

For example, imagine that you want an image that is the same as `core-image-minimal` but includes your `helloworld` program and the `strace` utility. You can do that with a two-line recipe file which includes (using the `require` keyword) the base image and adds the packages you want. It is conventional to put the image in a directory named `images`, so add the recipe `nova-image.bb` with this content in `meta-nova/recipes-local/images`:

```
require recipes-core/images/core-image-minimal.bb
IMAGE_INSTALL += "helloworld strace"
```

Now you can remove the `IMAGE_INSTALL`_append line from your `local.conf` and build it using:

```
$ bitbake nova-image
```

If you want to go further and take total control of the contents of the root filesystem, you can start from scratch with an empty `IMAGE_INSTALL` variable and populate it like this:

```
SUMMARY = "A small image with helloworld and strace packages"
IMAGE_INSTALL = "packagegroup-core-boot helloworld strace"
IMAGE_LINGUAS = " "
LICENSE = "MIT"
IMAGE_ROOTFS_SIZE ?= "8192"
inherit core-image
```

`IMAGE_LINGUAS` contains a list of `glibc` locales to be installed in the target image. They can take up a lot of space so, in this case, we set the list to be empty, which is fine so long as we do not need locale-dependent library functions. `IMAGE_ROOTFS_SIZE` is the size of the resulting disk image, in KiB. Most of the work is done by the `core-image` class which we inherit at the end.

Creating an SDK

It is very useful to be able to create a standalone toolchain that other developers can install, avoiding the need for everyone in the team to have a full installation of the Yocto Project. Ideally, you want the toolchain to include development libraries and header files for all the libraries installed on the target. You can do that for any image using the `populate_sdk` task, as shown here:

```
$ bitbake nova-image -c populate_sdk
```

The result is a self-installing shell script in `tmp/deploy/sdk` named:

```
poky-<c_library>-<host_machine>-<target_image><target_machine>-
toolchain-<version>.sh
```

Here is an example:

```
poky-glibc-x86_64-nova-image-cortexa8hf-vfp-neon-toolchain-  
1.8.1.sh
```

Note that, by default, the toolchain does not include static libraries. You can enable them individually by adding lines like this to your local.conf or the image recipe:

```
TOOLCHAIN_TARGET_TASK_append = " glibc-staticdev"
```

You can also enable them globally as shown:

```
SDKIMAGE_FEATURES_append = " staticdev-pkgs"
```

If you only want a basic toolchain with just C and C++ cross compilers, the C library and header files, you can instead run:

```
$ bitbake meta-toolchain
```

To install the SDK, just run the shell script. The default install directory is /opt/poky, but the install script allows you to change that:

```
$ tmp/deploy/sdk/poky-glibc-x86_64-nova-image-cortexa8hf-vfp-neon-  
toolchain-1.8.1.sh
```

```
Enter target directory for SDK (default: /opt/poky/1.8.1):
```

```
You are about to install the SDK to "/opt/poky/1.8.1". Proceed[Y/n]?
```

```
[sudo] password for chris:
```

```
Extracting SDK...done
```

```
Setting it up...done
```

```
SDK has been successfully set up and is ready to be used.
```

To make use of the toolchain, first source the environment set up script:

```
. /opt/poky/1.8.1/environment-setup-cortexa8hf-vfp-neon-poky-linux-  
gnueabi
```

Toolchains generated in this way are not configured with a valid sysroot:

```
$ arm-poky-linux-gnueabi-gcc -print-sysroot
```

```
/not/exist
```

Consequently, if you try to cross compile as I have shown in previous chapters, it will fail like this:

```
$ arm-poky-linux-gnueabi-gcc helloworld.c -o helloworld

helloworld.c:1:19: fatal error: stdio.h: No such file or directory
 #include <stdio.h>
 ^
compilation terminated.
```

This is because the compiler has been configured to be generic to a wide range of ARM processors, and the fine tuning is done when you launch it using the right set of `gcc` flags. So long as you use `$CC` to compile, everything should work fine:

```
$ $CC helloworld.c -o helloworld
```

License audit

The Yocto Project insists that each package has a license. A copy of the license is in `tmp/deploy/licenses/ [packagename.e]` for each package, as it is built. In addition, a summary of the packages and licenses used in an image are in the `<image name>-<machine name>-<date stamp>` directory. This is shown here:

```
$ ls tmp/deploy/licenses/nova-image-beaglebone-20151104150124
license.manifest package.manifest
```

The first file lists the licenses used by each package, the second lists the package names only.

Further reading

You can have look at the following documentation for more information:

- *The Buildroot User Manual*, <http://buildroot.org/downloads/manual/manual.html>
- *Yocto Project* documentation: there are nine reference guides plus a tenth which is a composite of the others (the so-called *Mega-manual*) at <https://www.yoctoproject.org/documentation>

- *Instant Buildroot*, by Daniel Manchón Vizuete, Packt Publishing, 2013
- *Embedded Linux Development with Yocto Project*, by Otavio Salvador and Daianne Angolini, Packt Publishing, 2014

Summary

Using a build system takes the hard work out of creating an embedded Linux system, and it is almost always better than hand crafting a roll your own system. There is a range of open source build systems available these days: Buildroot and the Yocto Project represent two different approaches. Buildroot is simple and quick, making it a good choice for fairly simple single-purpose devices: traditional embedded Linux as I like to think of them.

The Yocto Project is more complex and flexible. It is package based, meaning that you have the option to install a package manager and perform updates of individual packages in the field. The meta layer structure makes it easy to extend the metadata and indeed there is good support throughout the community and industry for the Yocto Project. The downside is that there is a very steep learning curve: you should expect it to take several months to become proficient with it, and even then it will sometimes do things that you didn't expect, or at least that is my experience.

Don't forget that any devices you create using these tools will need to be maintained in the field for a period of time, often many years. The Yocto Project will provide point releases for about one year after a release, Buildroot usually does not provide any point releases. In either case you will find yourself having to maintain your release yourself or else paying for commercial support. The third possibility, ignoring the problem, should not be considered an option!

In the next chapter I will look at file storage and filesystems, and at the way that the choices you make there will affect the stability and maintainability of your embedded Linux.

7

Creating a Storage Strategy

The mass storage options for embedded devices have a great impact on the rest of the system in terms of robustness, speed, and methods of in-field updates.

Most devices employ flash memory in some form or other. Flash memory has become much less expensive over the past few years as storage capacities have increased from tens of megabytes to tens of gigabytes.

In this chapter, I will begin with a detailed look at the technology behind flash memory and how different memory organization affects the low level driver software that has to manage it, including the Linux memory technology device layer, MTD.

For each flash technology, there are different choices of filesystem. I will describe those most commonly found on embedded devices and complete the survey with a section giving a summary of choices for each type of flash.

The last sections consider techniques to make the best use of flash memory, look at how to update devices in the field, and draw everything together into a coherent storage strategy.

Storage options

Embedded devices need storage that takes little power, is physically compact, robust, and reliable over a lifetime of perhaps tens of years. In almost all cases, that means solid state storage, which was introduced many years ago with **read-only memory (ROM)**, but for the past 20 years it has been flash memory of some kind. There have been several generations of flash memory in that time, progressing from NOR to NAND to managed flash such as eMMC.

NOR flash is expensive but reliable and can be mapped into the CPU address space, which allows you to execute code directly from flash. NOR flash chips are low-capacity, ranging from a few megabytes to a gigabyte or so.

NAND flash memory is much cheaper than NOR and is available in higher capacities, in the range of tens of megabytes to tens of gigabytes. However, it needs a lot of hardware and software support to turn it into a useful storage medium.

Managed flash memory consists of one or more NAND flash chips packaged with a controller which handles the complexities of flash memory and presents a hardware interface similar to that of a hard disk. The attraction is that it removes complexity from the driver software and insulates the system designer against the frequent changes in flash technology. SD cards, eMMC chips, and USB flash drives fit into this category. Almost all of the current generation of smartphones and tablets have eMMC storage, and that trend is likely to progress with other categories of embedded devices.

Hard drives are seldom found in embedded systems. One exception is digital video recording in set-top boxes and smart TVs in which a large amount of storage is needed with fast write times.

In all cases, robustness is of prime importance: you want the device to boot and reach a functional state despite power failures and unexpected resets. You should choose filesystems that behave well under such circumstances.

NOR flash

The memory cells in NOR flash chips are arranged into erase blocks of, for example, 128 KiB. Erasing a block sets all the bits to 1. It can be programmed one word at a time (8, 16 or 32 bits, depending on the data bus width). Each erase cycle damages the memory cells slightly and, after a number of cycles, the erase block becomes unreliable and cannot be used anymore. The maximum number of erase cycles should be given in the data sheet for the chip but is usually in the range of 100K to 1M.

The data can be read word by word. The chip is usually mapped into the CPU address space which means that you can execute code directly from NOR flash. This makes it a convenient place to put the bootloader code as it needs no initialization beyond hardwiring the address mapping. SoCs that support NOR flash in this way have configurations that give a default memory mapping such that it encompasses the reset vector of the CPU.

The kernel, and even the root filesystem, can also be located in flash memory, avoiding the need for copying them into RAM and thus creating devices with small memory footprints. The technique is known as **eXecute In Place**, or **XIP**. It is very specialized and I will not examine it further here. There are some references at the end of the chapter.

There is a standard register-level interface for NOR flash chips called the **common flash interface** or **CFI**, which all modern chips support.

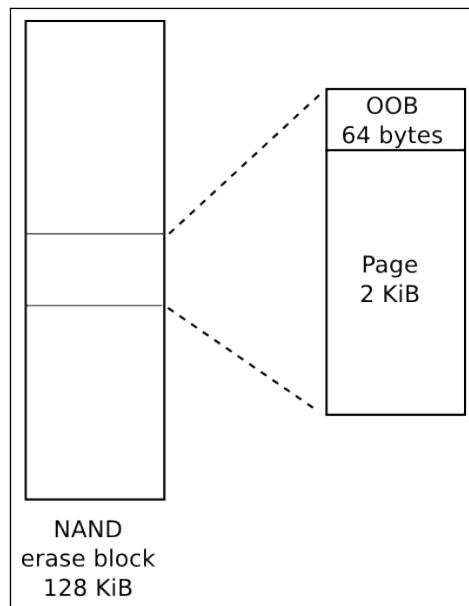
NAND flash

NAND flash is much cheaper than NOR flash and has a higher capacity. First generation NAND chips stored one bit per memory cell in what is now known as an **SLC** or **single level cell** organization. Later generations moved on to two bits per cell in **multi-level cell** (**MLC**) chips and now to three bits per cell in **tri-level cell** (**TLC**) chips. As the number of bits per cell has increased, the reliability of the storage has decreased, requiring more complex controller hardware and software to compensate.

As with NOR flash, NAND flash is organized into erase blocks ranging in size from 16 KiB to 512 KiB and, once again, erasing a block sets all the bits to 1. However, the number of erase cycles before the block becomes unreliable is lower, typically as few as 1K cycles for TLC chips and up to 100K for SLC. NAND flash can only be read and written in pages, usually of 2 or 4 KiB. Since they cannot be accessed byte-by-byte, they cannot be mapped into the address space and so code and data have to be copied into RAM before they can be accessed.

Data transfers to and from the chip are prone to bit flips, which can be detected and corrected using Error Correction Codes. SLC chips generally use a simple hamming code which can be implemented efficiently in software and can correct a single bit error in a page read. MLC and TLC chips need more sophisticated codes such as **BCH (Bose-Chaudhuri-Hocquenghem)** which can correct up to 8-bit errors per page. These need hardware support.

The error correction codes have to be stored somewhere and so there is an extra area of memory per page known as the **out of band (OOB)** area, or also the spare area. SLC designs usually have 1 byte of OOB per 32 bytes of main storage so, for a 2 KiB page device, the OOB is 64 bytes per page and for a 4 KiB page, 128 bytes. MLC and TLC chips have proportionally larger OOB areas to accommodate more complex error correction codes. The following diagram shows the organization of a chip with a 128 KiB erase block and 2 KiB pages:



During production, the manufacturer tests all the blocks and marks any that fail by setting a flag in the OOB area of each page in the block. It is not uncommon to find that brand new chips have up to 2% of their blocks marked bad in this way. Furthermore, it is within the specification for a similar proportion of blocks to give errors on erase before the erase cycle limit is reached. The NAND flash driver should detect this and mark it as bad.

After space has been taken in the OOB area for a bad block flag and ECC bytes, there are still some bytes left. Some flash filesystems make use of these free bytes to store filesystem metadata. Consequently, lots of people are interested in the layout of the OOB area: the SoC ROM boot code, the bootloader, the kernel MTD driver, the filesystem code, and the tools to create filesystem images. There is not much standardization so it is easy to get into a situation in which the bootloader writes data using an OOB format that cannot be read by the kernel MTD driver. It is up to you to make sure that they all agree.

Access to NAND flash chips requires a NAND flash controller, which is usually part of the SoC. You will need the corresponding driver in the bootloader and kernel. The NAND flash controller handles the hardware interface to the chip, transferring data to and from pages, and may include hardware for error correction.

There is a standard register-level interface for NAND flash chips known as the **open NAND flash interface** or **ONFi** which most modern chips adhere to. See <http://www.onfi.org>.

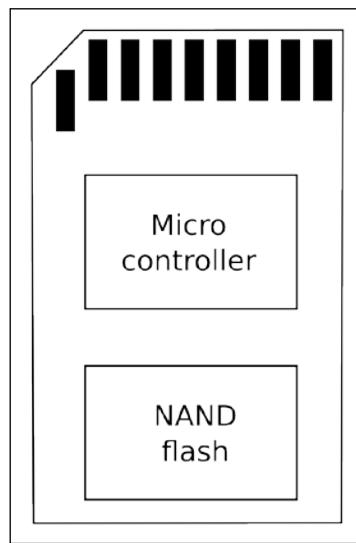
Managed flash

The burden of supporting flash memory in the operating system, NAND in particular, becomes less if there is a well-defined hardware interface and a standard flash controller that hides the complexities of the memory. This is managed flash memory and it is becoming more and more common. In essence, it means combining one or more flash chips with a micro controller that offers an ideal storage device with a small sector size that is compatible with conventional filesystems. The most important types of managed flash for embedded systems are **Secure Digital (SD)** cards and the embedded variant known as **eMMC**.

MultiMediaCard and secure digital cards

The **MultiMediaCard (MMC)** was introduced in 1997 by SanDisk and Siemens as a form of packaged storage using flash memory. Shortly after, in 1999, SanDisk, Matsushita, and Toshiba created the SD card which is based on MMC but adds encryption and DRM (that is the secure part). Both were intended for consumer electronics such as digital cameras, music players, and similar devices. Currently, SD cards are the dominant form of managed flash for consumer and embedded electronics, even though the encryption features are seldom used. Newer versions of the SD specification allow for smaller packaging (mini SD and micro SD which is often written as uSD) and larger capacities: high capacity, SDHC, up to 32 GB and extended capacity, SDXC, up to 2 TB.

The hardware interface for MMC and SD cards is very similar, and it is possible to use fully-sized MMC in full-sized SD card slots (but not the other way round). Early incarnations used a 1-bit **Serial Peripheral Interface (SPI)**; more recent cards use a 4-bit interface. There is a command set for reading and writing memory in sectors of 512 bytes. Inside the package there is a microcontroller and one or more NAND flash chips, as shown in the diagram that follows:



The microcontroller implements the command set and manages the flash memory, performing the function of a flash translation layer, as described later on in this chapter. They are pre-formatted with a FAT filesystem: FAT16 on SDSC cards, FAT32 on SDHC, and exFAT on SDXC. The quality of the NAND flash chips and the software on the microcontroller varies greatly between cards. It is questionable whether any of them are sufficiently reliable for deep embedded use, and certainly not with a FAT filesystem which is prone to file corruption. Remember that the prime use case for MMC and SD cards is for removable storage on cameras, tablets, and phones.

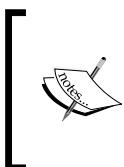
eMMC

eMMC or **Embedded MMC** is simply MMC memory packaged so that it can be soldered on to the motherboard, using a 4- or 8-bit interface for data transfer. However, they are intended to be used as storage for an operating system so the components are capable of performing that task. The chips are usually not pre-formatted with any filesystem.

Other types of managed flash

One of the first managed flash technologies was **CompactFlash (CF)**, using a subset of the **PCMCIA (Personal Computer Memory Card International Association)** interface. CF exposes the memory through a parallel ATA interface and appears to the operating system as a standard hard disk. They are common in x86-based single board computers and professional video and camera equipment.

One other format which we use every day is the USB flash drive. In this case, the memory is accessed through a USB interface and the controller implements the USB mass storage specification as well as the flash translation layer and interface to the flash chip or chips. The USB mass storage protocol, in turn, is based on the SCSI disk command set. As with MMC and SD cards, they are usually pre-formatted with a FAT filesystem. Their main use case in embedded systems is exchanging data with PCs.



A recent addition to the list of options for managed flash storage is **universal flash storage (UFS)**. Like eMMC, it is packaged in a chip that is mounted on the motherboard. It has a high-speed serial interface and can achieve data rates greater than eMMC. It supports a SCSI disk command set.

Accessing flash memory from the bootloader

In *Chapter 3, All About Bootloaders*, I mentioned the need for the bootloader to load kernel binaries and other images from various flash devices and to be able to perform system maintenance tasks such as erasing and reprogramming flash memory. It follows that the bootloader must have the drivers and infrastructure to support read, erase, and write operations on the type of memory you have, whether it be NOR, NAND, or managed. I will use U-Boot in the following example; other bootloaders follow a similar pattern.

U-Boot and NOR flash

U-Boot has drivers for NOR CFI chips in `drivers/mtd` and has the commands `erase` to erase memory and `cp.b` to copy data byte by byte, programming the flash. Suppose that you have NOR flash memory mapped from `0x40000000` to `0x48000000`, of which 4MiB starting at `0x40040000` is a kernel image, then you would load a new kernel into flash using these U-Boot commands:

```
U-Boot# tftpboot 100000 uImage
U-Boot# erase 40040000 403fffff
U-Boot# cp.b 100000 40040000 $(filesize)
```

The variable `filesize` in the preceding example is set by the `tftpboot` command to the size of the file just downloaded.

U-Boot and NAND flash

For NAND flash, you need a driver for the NAND flash controller on your SoC, which you can find in `drivers/mtd/nand`. You use the `nand` command to manage the memory using the sub-commands `erase`, `write`, and `read`. This example shows a kernel image being loaded into RAM at `0x82000000` and then placed into flash starting at offset `0x280000`:

```
U-Boot# tftpboot 82000000 uImage
U-Boot# nand erase 280000 400000
U-Boot# nand write 82000000 280000 $(filesize)
```

U-Boot can also read files stored in the JFFS2, YAFFS2, and UBIFS filesystems.

U-Boot and MMC, SD and eMMC

U-Boot has drivers for several MMC controllers in `drivers/mmc`. You can access the raw data using `mmc read` and `mmc write` at the user interface level, which allows you to handle raw kernel and filesystem images.

U-boot can also read files from the FAT32 and ext4 filesystems on MMC storage.

Accessing flash memory from Linux

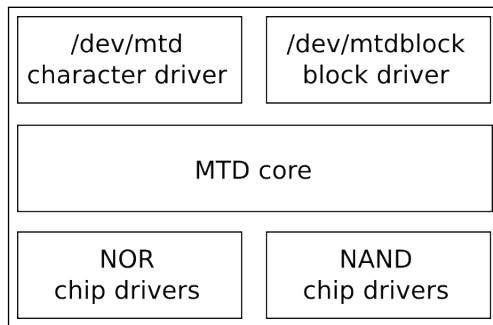
Raw NOR and NAND flash memory is handled by the memory technology device sub-system, or MTD, which provides basic interfaces to read, erase, and write blocks of flash memory. In the case of NAND flash, there are functions to handle the OOB area and to identify bad blocks.

For managed flash, you need drivers to handle the particular hardware interface. MMC/SD cards and eMMC use the `mmcblk` driver; CompactFlash and hard drives use the SCSI Disk driver, `sd`. USB flash drives use the `usb_storage` driver together with the `sd` driver.

Memory technology devices

The **memory technology devices (MTD)**, sub-system was started by David Woodhouse in 1999 and has been extensively developed over the intervening years. In this section, I will concentrate on the way it handles the two main technologies, NOR and NAND flash.

MTD consists of three layers: a core set of functions, a set of drivers for various types of chips, and user-level drivers that present the flash memory as a character device or a block device, as shown in the following diagram:



The chip drivers are at the lowest level and interface with flash chips. Only a small number of drivers are needed for NOR flash chips, enough to cover the CFI standard and variations plus a few non-compliant chips which are now mostly obsolete. For NAND flash, you will need a driver for the NAND flash controller you are using; this is usually supplied as part of the board support package. There are drivers for about 40 of them in the current mainline kernel in the directory `drivers/mtd/nand`.

MTD partitions

In most cases, you will want to partition the flash memory into a number of areas, for example, to provide space for a bootloader, a kernel image, or a root filesystem. In MTD, there are several ways to specify the size and location of partitions, the main ones being:

- Through the kernel command line using `CONFIG_MTD_CMDLINE_PARTS`
- Via the device tree using `CONFIG_MTD_OF_PARTS`
- With a platform mapping driver

In the case of the first option, the kernel command line option to use is `mtdparts`, which is defined as follows in the Linux source code in `drivers/mtd/cmdlinepart.c`:

```
mtdparts=<mtddef>[,<mtddef>
<mtddef>  := <mtd-id>:<partdef>[,<partdef>]
<mtd-id>  := unique name for the chip
<partdef> := <size>[@<offset>]<name>[ro][lk]
<size>    := size of partition OR "--" to denote all remaining
              space
<offset>  := offset to the start of the partition; leave blank
              to follow the previous partition without any gap
<name>    := '(' NAME ')'
```

Perhaps an example will help. Imagine that you have one flash chip of 128 MB that is to be divided into five partitions. A typical command line would be:

```
mtdparts=:512k(SPL)ro,780k(U-Boot)ro,128k(U-BootEnv),
4m(Kernel),-(Filesystem)
```

The first element, before the colon, `:`, is `mtd-id`, which identifies the flash chip, either by number or by the name assigned by the board support package. If there is only one chip, as here, it can be left empty. If there is more than one chip, the information for each is separated by a semicolon, `;`. Then, for each chip, there is a comma-separated list of partitions, each with a size in bytes, kilobytes, `k`, or megabytes, `m`, and a name in brackets. The `ro` suffix makes the partition read-only to MTD and is often used to prevent accidental overwriting of the bootloader. The size of the last partition for the chip may be replaced by a dash, `-`, indicating that it should take up all the remaining space.

You can see a summary of the configuration at runtime by reading `/proc/mtd`:

```
# cat /proc/mtd
dev:      size   erasesize   name
mtd0: 00080000 00020000 "SPL"
mtd1: 000C3000 00020000 "U-Boot"
mtd2: 00020000 00020000 "U-BootEnv"
mtd3: 00400000 00020000 "Kernel"
mtd4: 07A9D000 00020000 "Filesystem"
```

There is more detailed information for each partition in `/sys/class/mtd`, including the erase block size and the page size, and it is nicely summarized using `mtdinfo`:

```
# mtdinfo /dev/mtd0
mtd0
  Name:                      SPL
  Type:                      nand
```

```
Eraseblock size:          131072 bytes, 128.0 KiB
Amount of eraseblocks:    4 (524288 bytes, 512.0 KiB)
Minimum input/output unit size: 2048 bytes
Sub-page size:           512 bytes
OOB size:                64 bytes
Character device major/minor: 90:0
Bad blocks are allowed:  true
Device is writable:      false
```

The equivalent partition information can be written as part of the device tree like so:

```
nand@0,0 {
    #address-cells = <1>;
    #size-cells = <1>;
    partition@0 {
        label = "SPL";
        reg = <0 0x80000>;
    };
    partition@80000 {
        label = "U-Boot";
        reg = <0x80000 0xc3000>;
    };
    partition@143000 {
        label = "U-BootEnv";
        reg = <0x143000 0x20000>;
    };
    partition@163000 {
        label = "Kernel";
        reg = <0x163000 0x400000>;
    };
    partition@563000 {
        label = "Filesystem";
        reg = <0x563000 0x7a9d000>;
    };
};
```

A third alternative is to code the partition information as platform data in an `mtd_partition` structure, as shown in this example taken from `arch/arm/mach-omap2/board-omap3beagle.c` (`NAND_BLOCK_SIZE` is defined elsewhere to be 128K):

```
static struct mtd_partition omap3beagle_nand_partitions[] = {
{
    .name         = "X-Loader",
```

```
.offset      = 0,
.size       = 4 * NAND_BLOCK_SIZE,
.mask_flags = MTD_WRITEABLE, /* force read-only */
},
{
    .name      = "U-Boot",
    .offset    = 0x80000;
    .size     = 15 * NAND_BLOCK_SIZE,
    .mask_flags = MTD_WRITEABLE, /* force read-only */
},
{
    .name      = "U-Boot Env",
    .offset    = 0x260000;
    .size     = 1 * NAND_BLOCK_SIZE,
},
{
    .name      = "Kernel",
    .offset    = 0x280000;
    .size     = 32 * NAND_BLOCK_SIZE,
},
{
    .name      = "File System",
    .offset    = 0x680000;
    .size     = MTDPART_SIZ_FULL,
},
};
```

MTD device drivers

The upper level of the MTD sub-system is a pair of device drivers:

- A character device, with a major number of 90. There are two device nodes per MTD partition number, N : `/dev/mtdN` (*minor number*= N^2) and `/dev/mtdNro` (*minor number*= $(N^2 + 1)$). The latter is just a read-only version of the former.
- A block device, with a major number of 31 and a minor number of N . The device nodes are in the form `/dev/mtdblockN`.

The MTD character device, mtd

The character devices are the most important: they allow you to access the underlying flash memory as an array of bytes so that you can read and write (program) the flash. It also implements a number of `ioctl` functions that allow you to erase blocks and to manage the OOB area on NAND chips. The following list is in `include/uapi/mtd/mtd-abi.h`:

IOCTL	Description
MEMGETINFO	Gets basic MTD characteristic information
MEMERASE	Erases blocks in the MTD partition
MEMWRITEOOB	Writes out-of-band data for the page
MEMREADOOB	Reads out-of-band data for the page
MEMLOCK	Locks the chip (if supported)
MEMUNLOCK	Unlocks the chip (if supported)
MEMGETREGIONCOUNT	Gets the number of erase regions: non-zero if there are erase blocks of differing sizes in the partition, which is common for NOR flash, rare on NAND
MEMGETREGIONINFO	If <code>MEMGETREGIONCOUNT</code> is non-zero, this can be used to get the offset, size, and block count of each region
MEMGETOOBSEL	Deprecated
MEMGETBADBLOCK	Gets the bad block flag
MEMSETBADBLOCK	Sets the bad block flag
OTPSELECT	Sets OTP (one-time programmable) mode, if the chip supports it
OTPGETREGIONCOUNT	Gets the number of OTP regions
OTPGETREGIONINFO	Gets information about an OTP region
ECCGETLAYOUT	Deprecated

There is a set of utility programs known as `mtd-utils` for manipulating flash memory that makes use of these `ioctl` functions. The source is available from <http://git.infradead.org/mtd-utils.git> and is available as a package in the Yocto Project and Buildroot. The essential tools are shown in the following list. The package also contains utilities for the JFFS2 and UBI/UBIFS filesystems which I will cover later. For each of these tools, the MTD character device is one of the parameters:

- **flash_erase**: Erases a range of blocks.
- **flash_lock**: Locks a range of blocks.
- **flash_unlock**: Unlocks a range of blocks.

- **nanddump**: Dumps memory from NAND flash, optionally including the OOB area. Skips bad blocks.
- **nandtest**: Tests and diagnostics for NAND flash.
- **nandwrite**: Writes (program) from a data file to NAND flash, skipping bad blocks.

 You must always erase flash memory before writing new contents to it: `flash_erase` is the command to do that.

To program NOR flash, you simply copy bytes to the MTD device node using the `cp` command or similar.

Unfortunately, this doesn't work with NAND memory as the copy will fail at the first bad block. Instead, use `nandwrite`, which skips over any bad blocks. To read back NAND memory, you should use `nanddump` which also skips bad blocks.

The MTD block device, `mtdblock`

The `mtdblock` driver is little used. Its purpose is to present flash memory as a block device which you can use to format and mount as a filesystem. However, it has severe limitations because it does not handle bad blocks in NAND flash, does not do wear leveling, and does not handle the mismatch in size between filesystem blocks and flash erase blocks. In other words, it does not have a flash translation layer, which is essential for reliable file storage. The only case where the `mtdblock` device is useful is to mount read-only filesystems such as Squashfs on top of reliable flash memory such as NOR.

 If you want a read-only filesystem on NAND flash, you should use the UBI driver, as described later in this chapter.

Logging kernel oops to MTD

Kernel errors, or oopsies, are normally logged via the `klogd` and `syslogd` daemons to a circular memory buffer or a file. Following a reboot, the log will be lost in the case of a ring buffer and, even in the case of a file, it may not have been properly written before the system crashed.



A more reliable method is to write oops and kernel panics to an MTD partition as a circular log buffer. You enable it with `CONFIG_MTD_OOPS` and you add `console=ttyMTDN` to the kernel command line, `N` being the MTD device number to write the messages to.

Simulating NAND memory

The NAND simulator emulates a NAND chip using system RAM. The main use is for testing code that has to be NAND-aware without access to physical NAND memory. In particular, the ability to simulate bad blocks, bit flips, and other errors allows you to test code paths that are difficult to exercise using real flash memory. For more information, the best place to look is in the code itself, which has a comprehensive description of the ways you can configure the driver. The code is in `drivers/mtd/nand/nandsim.c`. Enable it with the kernel configuration `CONFIG_MTD_NAND_NANDSIM`.

The MMC block driver

MMC/SD cards and eMMC chips are accessed using the `mmcblk` block driver. You need a host controller to match the MMC adapter you are using, which is part of the board support package. The drivers are located in the Linux source code in `drivers/mmc/host`.

MMC storage is partitioned using a partition table in exactly the same way you would for hard disks, using `fdisk` or a similar utility.

Filesystems for flash memory

There are several challenges when making efficient use of flash memory for mass storage: the mismatch between the size of an erase block and a disk sector, the limited number of erase cycles per erase block, and the need for bad block handling on NAND chips. These differences are resolved by a **Flash Translation Layer globally**, or **FTL**.

Flash translation layers

A flash translation layer has the following features:

- **Sub allocation:** Filesystems work best with a small allocation unit, traditionally a 512-byte sector. This is much smaller than a flash erase block of 128 KiB or more. Therefore erase blocks have to be sub-divided into smaller units to avoid wasting large amounts of space.

- **Garbage collection:** A consequence of sub-allocation is that an erase block will contain a mixture of good data and stale data after the filesystem has been in use for a while. Since we can only free up whole erase blocks, the only way to reclaim the free space is to coalesce the good data into one place and return the now empty erase block to the free list: this is garbage collection, and is usually implemented as a background thread.
- **Wear leveling:** There is a limit on the number of erase cycles for each block. To maximize the lifespan of a chip, it is important to move data around so that each block is erased roughly the same number of times.
- **Bad block handling:** On NAND flash chips, you have to avoid using any block marked bad and also mark good blocks as bad if they cannot be erased.
- **Robustness:** Embedded devices may be powered off or reset without warning, so any filesystem should be able to cope without corruption, usually by incorporating a journal or log of transactions.

There are several ways to deploy the flash translation layer:

- **In the filesystem:** as with JFFS2, YAFFS2, and UBIFS
- **In the block device driver:** the UBI driver, on which UBIFS depends, implements some aspects of a flash translation layer
- **In the device controller:** as with managed flash devices

When the flash translation layer is in the filesystem or the block driver, the code is part of the kernel and so it is open source, meaning that we can see how it works and we can expect that it will be improved over time. On the other hand, the FTL is inside a managed flash device; it is hidden from view and we cannot verify whether or not it works as we would want. Not only that, but putting the FTL into the disk controller means that it misses out on information that is held at the filesystem layer such as which sectors belong to files that have been deleted and so do not contain useful data anymore. The latter problem is solved by adding commands that pass this information between the filesystem and the device I will describe in the section on the TRIM command later, but the question of code visibility remains. If you are using managed flash, you just have to choose a manufacturer you can trust.

Filesystems for NOR and NAND flash memory

To use raw flash chips for mass storage, you have to use a filesystem that understands the peculiarities of the underlying technology. There are three such filesystems:

- **Journaling Flash File System 2, JFFS2:** This was the first flash filesystem for Linux, and is still in use today. It works for NOR and NAND memory, but is notoriously slow during mount.
- **Yet Another Flash File System 2, YAFFS2:** This is similar to JFFS2, but specifically for NAND flash memory. It was adopted by Google as the preferred raw flash filesystem on Android devices.
- **Unsorted Block Image File System, UBIFS:** This is the latest flash-aware filesystem for both NOR and NAND memory, which is used in conjunction with the UBI block driver. It generally offers better performance than JFFS2 or YAFFS2, and so should be the preferred solution for new designs.

All of these use MTD as the common interface to flash memory.

JFFS2

The Journaling Flash File System had its beginnings in the software for the Axis 2100 network camera in 1999. For many years, it was the only flash filesystem for Linux and has been deployed on many thousands of different types of devices. Today, it is not the best choice, but I will cover it first because it shows the beginning of the evolutionary path.

JFFS2 is a log-structured filesystem that uses MTD to access flash memory. In a log-structured filesystem, changes are written sequentially as nodes to the flash memory. A node may contain changes to a directory, such as the names of files created and deleted, or it may contain changes to file data. After a while, a node may be superseded by information contained in subsequent nodes and becomes an obsolete node.

Erase blocks are categorized into three types:

- **free:** It contains no nodes at all
- **clean:** It contains only valid nodes
- **dirty:** It contains at least one obsolete node

At any one time, there is one block receiving updates which is called the open block. If power is lost or the system is reset, the only data that can be lost is the last write to the open block. In addition, nodes are compressed as they are written, increasing the effective storage capacity of the flash chip, which is important if you are using expensive NOR flash memory.

When the number of free blocks falls below a threshold, a garbage collector kernel thread is started, which scans for dirty blocks and copies the valid nodes into the open block, and then frees up the dirty block.

At the same time, the garbage collector provides a crude form of wear leveling because it cycles valid data from one block to another. The way that the open block is chosen means that each block is erased roughly the same number of times so long as it contains data that changes from time to time. Sometimes a clean block is chosen for garbage collection to make sure that blocks containing static data that is seldom written are also wear leveled.

JFFS2 filesystems have a write through cache, meaning that writes are written to the flash memory synchronously as if they have been mounted with a `-o sync` option. While improving reliability, it does increase the time to write data. There is a further problem with small writes: if the length of a write is comparable to the size of the node header (40 bytes) the overhead becomes high. A well-known corner case is log files, produced, for example, by syslogd.

Summary nodes

There is one overriding disadvantage to JFFS2: since there is no on-chip index, the directory structure has to be deduced at mount-time by reading the log from start to finish. At the end of the scan, you have a complete picture of the directory structure of the valid nodes, but the time taken is proportional to the size of the partition. It is not uncommon to see mount times of the order of one second per megabyte, leading to total mount times of tens or hundreds of seconds.

To reduce the time to scan during mount, summary nodes became an option in Linux 2.6.15. A summary node is written at the end of the open erase block just before it is closed. The summary node contains all of the information needed for the mount-time scan, thereby reducing the amount of data to process during the scan. Summary nodes can reduce mount times by a factor of between two and five, at the expense of an overhead of about 5% of the storage space. They are enabled with the kernel configuration `CONFIG_JFFS2_SUMMARY`.

Clean markers

An erased block with all bits set to 1 is indistinguishable from a block that has been written with 1's, but the latter has not had its memory cells refreshed and cannot be programmed again until it is erased. JFFS2 uses a mechanism called clean markers to distinguish between these two situations. After a successful block erase, a clean marker is written, either to the beginning of the block, or to the OOB area of the first page of the block. If the clean marker exists then it must be a clean block.

Creating a JFFS2 filesystem

Creating an empty JFFS2 filesystem at runtime is as simple as erasing an MTD partition with clean markers and then mounting it. There is no formatting step because a blank JFFS2 filesystem consists entirely of free blocks. For example, to format MTD partition 6, you would enter these commands on the device:

```
# flash_erase -j /dev/mtd6 0 0
# mount -t jffs2 mtd6 /mnt
```

The `-j` option to `flash_erase` adds the clean markers, and mounting with type `jffs2` presents the partition as an empty filesystem. Note that the device to be mounted is given as `mtd6`, not `/dev/mtd6`. Alternatively, you can give the block device node `/dev/mtdblock6`. This is just a peculiarity of JFFS2. Once mounted, you can treat it like any filesystem and, when you next boot and mount it, all the files will still be there.

You can create a filesystem image directly from the staging area of your development system using `mkfs.jffs2` to write out the files in JFFS2 format and `sumtool` to add the summary nodes. Both of these are part of the `mtd-utils` package.

As an example, to create an image of the files in `rootfs` for a NAND flash device with an erase block size of 128 KB (0x20000) and with summary nodes, you would use these two commands:

```
$ mkfs.jffs2 -n -e 0x20000 -p -d ~/rootfs -o ~/rootfs.jffs2
$ sumtool -n -e 0x20000 -p -i ~/rootfs.jffs2 -o ~/rootfs-sum.jffs2
```

The `-p` option adds padding at the end of the image file to make it a whole number of erase blocks. The `-n` option suppresses the creation of clean markers in the image, which is normal for NAND devices as the clean marker is in the OOB area. For NOR devices, you would leave out the `-n` option. You can use a device table with `mkfs.jffs2` to set the permissions and the ownership of files by adding `-D [device table]`. Of course, Buildroot and the Yocto Project will do all this for you.

You can program the image into flash memory from your bootloader. For example, if you have loaded a filesystem image into RAM at address 0x82000000 and you want to load it into a flash partition begins at 0x163000 bytes from the start of the flash chip and is 0x7a9d000 bytes long, the U-Boot commands would be:

```
nand erase clean 163000 7a9d000
nand write 82000000 163000 7a9d000
```

You can do the same thing from Linux using the mtd driver like this:

```
# flash_erase -j /dev/mtd6 0 0
# nandwrite /dev/mtd6 rootfs-sum.jffs2
```

To boot with a JFFS2 root filesystem, you need to pass the `mtdblock` device on the kernel command line for the partition and a root `fstype` because JFFS2 cannot be auto-detected:

```
root=/dev/mtdblock6 rootfstype=jffs2
```

YAFFS2

The YAFFS filesystem was written by Charles Manning beginning in 2001, specifically to handle NAND flash chips at a time when JFFS2 did not. Subsequent changes to handle larger (2 KiB) page sizes resulted in YAFFS2. The website for YAFFS is <http://www.yaffs.net>.

YAFFS is also a log-structured filesystem following the same design principles as JFFS2. The different design decisions mean that it has a faster mount-time scan, simpler and faster garbage collection, and has no compression, which speeds up reads and writes at the expense of less efficient use of storage.

YAFFS is not limited to Linux; it has been ported to a wide range of operating systems. It has a dual license: GPLv2 to be compatible with Linux, and a commercial license for other operating systems. Unfortunately, the YAFFS code has never been merged into mainline Linux so you will have to patch your kernel, as shown in the following code.

To get YAFFS2 and patch a kernel, you would:

```
$ git clone git://www.aleph1.co.uk/yaffs2
$ cd yaffs2
$ ./patch-ker.sh c m <path to your link source>
```

Then, configure the kernel with `CONFIG_YAFFS_YAFFS2`.

Creating a YAFFS2 filesystem

As with JFFS2, to create a YAFFS2 filesystem at runtime, you only need to erase the partition and mount it but note that, in this case, you do not enable clean markers:

```
# flash_erase /dev/mtd/mtd6 0 0
# mount -t yaffs2 /dev/mtdblock6 /mnt
```

To create a filesystem image, the simplest thing to do is use the `mkyaffs2` tool from <https://code.google.com/p/yaffs2utils> using the following command:

```
$ mkyaffs2 -c 2048 -s 64 rootfs rootfs.yaffs2
```

Here `-c` is the page size and `-s` the OOB size. There is a tool named `mkyaffs2image` that is part of the YAFFS code, but it has a couple of drawbacks. Firstly, the page and OOB size are hard-coded in the source: you will have to edit and recompile if you have memory that does not match the defaults of 2,048 and 64. Secondly, the OOB layout is incompatible with MTD, which uses the first two bytes as a bad block marker, whereas `mkyaffs2image` uses those bytes to store part of the YAFFS metadata.

To copy the image to the MTD partition from a Linux shell prompt, follow these steps:

```
# flash_erase /dev/mtd6 0 0
# nandwrite -a /dev/mtd6 rootfs.yaffs2
```

To boot with a YAFFS2 root filesystem, add the following to the kernel command line:

```
root=/dev/mtdblock6 rootfstype=yaffs2
```

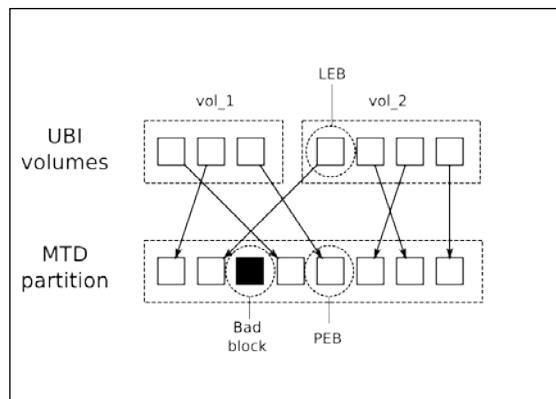
UBI and UBIFS

The **unsorted block image (UBI)** driver, is a volume manager for flash memory which takes care of bad block handling and wear leveling. It was implemented by Artem Bityutskiy and first appeared in Linux 2.6.22. In parallel with that, engineers at Nokia were working on a filesystem that would take advantage of the features of UBI which they called UBIFS; it appeared in Linux 2.6.27. Splitting the flash translation layer in this way makes the code more modular and also allows other filesystems to take advantage of the UBI driver, as we shall see later on.

UBI

UBI provides an idealized, reliable view of a flash chip by mapping **physical erase blocks (PEB)** to **logical erase blocks (LEB)**. Bad blocks are not mapped to LEBs and so are never used. If a block cannot be erased, it is marked as bad and dropped from the mapping. UBI keeps a count of the number of times each PEB has been erased in the header of the LEB and changes the mapping to ensure that each PEB is erased the same number of times.

UBI accesses the flash memory through the MTD layer. As an extra feature, it can divide an MTD partition into a number of UBI volumes, which improves wear leveling in the following way. Imagine that you have two filesystems, one containing fairly static data, for example, a root filesystem, and the other containing data that is constantly changing. If they are stored in separate MTD partitions, the wear leveling only has an effect on the second one, whereas, if you choose to store them in two UBI volumes in a single MTD partition, the wear leveling takes place over both areas of the storage and the lifetime of the flash memory is increased. The following diagram illustrates this situation:



In this way, UBI fulfills two of the requirements of a flash translation layer: wear leveling and bad block handling.

To prepare an MTD partition for UBI, you don't use `flash_erase` as with JFFS2 and YAFFS2, instead you use the `ubiformat` utility, which preserves the erase counts that are stored in the `_PEB_` headers. `ubiformat` needs to know the minimum unit of IO which, for most NAND flash chips, is the page size, but some chips allow reading and writing in sub pages that are a half or a quarter of the page size. Consult the chip data sheet for details and, if in doubt, use the page size. This example prepares `mtd6` using a page size of 2,048 bytes:

```
# ubiformat /dev/mtd6 -s 2048
```

You use the `ubiattach` command to load the UBI driver on an MTD partition that has been prepared in this way:

```
# ubiattach -p /dev/mtd6 -O 2048
```

This creates the device node `/dev/ubi0` through which you can access the UBI volumes. You can use `ubiattach` multiple times for other MTD partitions, in which case they can be accessed through `/dev/ubi1`, `/dev/ubi2`, and so on.

The PEB to LEB mapping is loaded into memory during the attach phase, a process that takes time proportional to the number of PEBs, typically a few seconds. A new feature was added in Linux 3.7 called the UBI fastmap which checkpoints the mapping to flash from time to time and so reduces the attach time. The kernel configuration option is `CONFIG_MTD_UBI_FASTMAP`.

The first time you attach to an MTD partition after a `ubiformat` there will be no volumes. You can create volumes using `ubimkvol`. For example, suppose you have a 128MB MTD partition and you want to split it into two volumes of 32 MB and 96 MB using a chip with 128 KB erase blocks and 2 KB pages:

```
# ubimkvol /dev/ubi0 -N vol_1 -s 32MiB
# ubimkvol /dev/ubi0 -N vol_2 -s 96MiB
```

Now, you have device the nodes `/dev/ubi0_0` and `/dev/ubi0_1`. You can confirm the situation using `ubinfo`:

```
# ubinfo -a /dev/ubi0
ubi0
Volumes count: 2
Logical eraseblock size: 15360 bytes, 15.0 KiB
Total amount of logical eraseblocks: 8192 (125829120 bytes,
120.0 MiB)
Amount of available logical eraseblocks: 0 (0 bytes)
Maximum count of volumes 89
Count of bad physical eraseblocks: 0
Count of reserved physical eraseblocks: 160
Current maximum erase counter value: 1
Minimum input/output unit size: 512 bytes
Character device major/minor: 250:0
Present volumes: 0, 1
Volume ID: 0 (on ubi0)
Type: dynamic
Alignment: 1
Size: 2185 LEBs (33561600 bytes, 32.0 MiB)
State: OK
Name: vol_1
Character device major/minor: 250:1
-----
Volume ID: 1 (on ubi0)
Type: dynamic
Alignment: 1
Size: 5843 LEBs (89748480 bytes, 85.6 MiB)
State: OK
Name: vol_2
Character device major/minor: 250:2
```

Note that, since each LEB has a header to contain the meta information used by UBI, the LEB is smaller than the PEB by one page. For example, a chip with a PEB size of 128 KB and 2 KB pages would have an LEB of 126 KB. This is important information that you will need when creating a UBIFS image.

UBIFS

UBIFS uses a UBI volume to create a robust filesystem. It adds sub-allocation and garbage collection to create a complete flash translation layer. Unlike JFFS2 and YAFFS2, it stores index information on-chip and so mounting is fast, although don't forget that attaching the UBI volume beforehand may take a significant amount of time. It also allows for write-back caching like a normal disk filesystem, which means that writes are much faster, but with the usual problem of potential loss of data that has not been flushed from the cache to flash memory in the event of power down. You can resolve the problem by making careful use of the `fsync(2)` and `fdatasync(2)` functions to force a flush of file data at crucial points.

UBIFS has a journal for fast recovery in the event power down. The journal takes up some space, typically 4 MiB or more, so UBIFS is not suitable for very small flash devices.

Once you have created the UBI volumes, you can mount them using the device node for the volume, `/dev/ubi0_0`, or by using the device node for the whole partition plus the volume name, as shown here:

```
# mount -t ubifs ubi0:vol_1 /mnt
```

Creating a filesystem image for UBIFS is a two-stage process: first you create a UBIFS image using `mkfs.ubifs`, and then embed it into a UBI volume using `ubinize`.

For the first stage, `mkfs.ubifs` needs to be informed of the page size with `-m`, the size of the UBI LEB with `-e`, remembering that the LEB is usually one page shorter than the PEB, and the maximum number of erase blocks in the volume with `-c`. If the first volume is 32 MiB and an erase block is 128 KiB, then the number of erase blocks is 256. So, to take the contents of the directory `rootfs` and create a UBIFS image named `rootfs.ubi`, you would type the following:

```
$ mkfs.ubifs -r rootfs -m 2048 -e 126KiB -c 256 -o rootfs.ubi
```

The second stage requires you to create a configuration file for `ubinize` which describes the characteristics of each volume in the image. The help page (`ubinize -h`) gives details of the format. This example creates two volumes, `vol_1` and `vol_2`:

```
[ubifsi_vol_1]
mode=ubi
image=rootfs.ubi
```

```
vol_id=0
vol_name=vol_1
vol_size=32MiB
vol_type=dynamic

[ubifs.vol_2]
mode=ubi
image=data.ubi
vol_id=1
vol_name=vol_2
vol_type=dynamic
vol_flags=autoresize
```

The second volume has an auto-resize flag and so will expand to fill the remaining space on the MTD partition. Only one volume can have this flag. From this information, ubinize will create an image file named by the `-o` parameter, with the PEB size `-p`, the page size `-m`, and the sub-page size `-s`:

```
$ ubinize -o ~/ubi.img -p 128KiB -m 2048 -s 512 ubinize.cfg
```

To install this image on the target, you would enter these commands on the target:

```
# ubiformat /dev/mtd6 -s 2048
# nandwrite /dev/mtd6 /ubi.img
# ubiattach -p /dev/mtd6 -O 2048
```

If you want to boot with a UBIFS root filesystem, you would give these kernel command line parameters:

```
ubi.mtd=6 root=ubi0:vol_1 rootfstype=ubifs
```

Filesystems for managed flash

As the trend towards managed flash technologies continues, particularly eMMC, we need to consider how to use it effectively. While they appear to have the same characteristics as hard disk drives, some NAND flash chips have the limitations of large erase blocks with limited erase cycles and bad block handling. And, of course, we need robustness in the event of losing power.

It is possible to use any of the normal disk filesystems but we should try to choose one that reduces disk writes and has a fast restart after an unscheduled shutdown, typically provided by a journal.

Flashbench

To make optimum use of the underlying flash memory, you need to know the erase block size and page size. Manufacturers do not publish these numbers, as a rule, but it is possible to deduce them by observing the behavior of the chip or card.

Flashbench is one such tool. It was initially written by Arnd Bergman, as described in the LWN article available at <http://lwn.net/Articles/428584>. You can get the code from <https://github.com/bradfa/flashbench>.

Here is a typical run on a SanDisk GiB SDHC card:

```
$ sudo ./flashbench -a /dev/mmcblk0 --blocksize=1024
align 536870912 pre 4.38ms on 4.48ms post 3.92ms diff 332µs
align 268435456 pre 4.86ms on 4.9ms post 4.48ms diff 227µs
align 134217728 pre 4.57ms on 5.99ms post 5.12ms diff 1.15ms
align 67108864 pre 4.95ms on 5.03ms post 4.54ms diff 292µs
align 33554432 pre 5.46ms on 5.48ms post 4.58ms diff 462µs
align 16777216 pre 3.16ms on 3.28ms post 2.52ms diff 446µs
align 8388608 pre 3.89ms on 4.1ms post 3.07ms diff 622µs
align 4194304 pre 4.01ms on 4.89ms post 3.9ms diff 940µs
align 2097152 pre 3.55ms on 4.42ms post 3.46ms diff 917µs
align 1048576 pre 4.19ms on 5.02ms post 4.09ms diff 876µs
align 524288 pre 3.83ms on 4.55ms post 3.65ms diff 805µs
align 262144 pre 3.95ms on 4.25ms post 3.57ms diff 485µs
align 131072 pre 4.2ms on 4.25ms post 3.58ms diff 362µs
align 65536 pre 3.89ms on 4.24ms post 3.57ms diff 511µs
align 32768 pre 3.94ms on 4.28ms post 3.6ms diff 502µs
align 16384 pre 4.82ms on 4.86ms post 4.17ms diff 372µs
align 8192 pre 4.81ms on 4.83ms post 4.16ms diff 349µs
align 4096 pre 4.16ms on 4.21ms post 4.16ms diff 52.4µs
align 2048 pre 4.16ms on 4.16ms post 4.17ms diff 9ns
```

Flashbench reads blocks of, in this case, 1,024 bytes just before and just after various power-of-two boundaries. As you cross a page or erase block boundary, the reads after the boundary take longer. The rightmost column shows the difference and is the one that is most interesting. Reading from the bottom, there is a big jump at 4 KiB, which is the most likely size of a page. There is a second jump from 52.4µs to 349µs at 8 KiB. This is fairly common and indicates that the card can use multi-plane accesses to read two 4 KiB pages at the same time. Beyond that, the differences are less well marked, but there is a clear jump from 485µs to 805µs at 512 KiB, which is probably the erase block size. Given that the card being tested is quite old, these are the sort of numbers you would expect.

Discard and TRIM

Usually, when you delete a file, only the modified directory node is written to storage while the sectors containing the file's contents remain unchanged. When the flash translation layer is in the disk controller, as with managed flash, it does not know that this group of disk sectors no longer contains useful data and so it ends up copying stale data.

In the last few years, the addition of transactions that pass information about deleted sectors down to the disk controller has improved the situation. The SCSI and SATA specifications have a **TRIM** command and MMC has a similar command named **ERASE**. In Linux, this feature is known as **discard**.

To make use of **discard**, you need a storage device that supports it – most current eMMC chips do – and a Linux device driver to match. You can check by looking at the block system queue parameters in `/sys/block/<block device>/queue/`. The ones of interest are as follows:

- `discard_granularity`: The size of the internal allocation unit of the device
- `discard_max_bytes`: The maximum number of bytes that can be discarded in one go
- `discard_zeroes_data`: If 1, discarded data will be set to zero

If the device or the device driver does not support **discard**, these values are all set to zero. These are the parameters you will see from the eMMC chip on the BeagleBone Black:

```
# grep -s "" /sys/block/mmcblk0/queue/discard_*
/sys/block/mmcblk0/queue/discard_granularity:2097152
/sys/block/mmcblk0/queue/discard_max_bytes:2199023255040
/sys/block/mmcblk0/queue/discard_zeroes_data:1
```

There is more information in the kernel documentation file, `Documentation/block/queue-sysfs.txt`.

You can enable **discard** when mounting a filesystem by adding the option `-o discard` to the `mount` command. Both ext4 and F2FS support it.



Make sure that the storage device supports **discard** before using the `-o discard` mount option, otherwise data loss can occur.

It is also possible to force `discard` from the command line independently of how the partition is mounted, using the `fstrim` command which is part of the `util-linux` package. Typically, you would run this command periodically, once a week perhaps, to free up unused space. `fstrim` operates on a mounted filesystem so, to trim the root filesystem `/`, you would type the following:

```
# fstrim -v /
/: 2061000704 bytes were trimmed
```

The preceding example uses the verbose option, `-v`, so that it prints out the number of bytes potentially freed up. In this case 2,061,000,704 is the approximate amount of free space in the filesystem, so it is the maximum amount of storage that could have been freed.

Ext4

The extended filesystem, `ext`, has been the main filesystem for Linux desktops since 1992. The current version, `ext4`, is very stable, well tested and has a journal that makes recovery from an unscheduled shutdown fast and mostly painless. It is a good choice for managed flash devices and you will find that it is the preferred filesystem for Android devices that have eMMC storage. If the device supports `discard`, you should mount with the option `-o discard`.

To format and create an `ext4` filesystem at runtime, you would type the following:

```
# mkfs.ext4 /dev/mmcblk0p2
# mount -t ext4 -o discard /dev/mmcblk0p1 /mnt
```

To create a filesystem image, you can use the `genext2fs` utility, available from <http://genext2fs.sourceforge.net>. In this example, I have specified the block size with `-B` and the number of blocks in the image with `-b`:

```
$ genext2fs -B 1024 -b 10000 -d rootfs rootfs.ext4
```

`genext2fs` can make use of a device table to set the file permissions and ownership, as described in *Chapter 5, Building a Root Filesystem*, with `-D [file table]`.

As the name implies, this will actually generate an image in `.ext2` format. You can upgrade using `tune2fs` as follows (details of the command options are in the main page for `tune2fs`):

```
$ tune2fs -j -J size=1 -O filetype,extents,uninit_bg,dir_index
rootfs.ext4
$ e2fsck -pDf rootfs.ext4
```

Both the Yocto Project and Buildroot use exactly these steps when creating images in .ext4 format.

While a journal is an asset for devices that may power down without warning, it does add extra write cycles to each write transaction, wearing out the flash memory. If the device is battery-powered, especially if the battery is not removable, the chances of an unscheduled power down are small and so you may want to leave the journal out.

F2FS

The **Flash-Friendly File System, F2FS**, is a log-structured filesystem designed for managed flash devices, especially eMMC and SD. It was written by Samsung and was merged into mainline Linux in 3.8. It is marked experimental, indicating that it has not been extensively deployed as yet, but it seems that some Android devices are using it.

F2FS takes into account the page and erase block sizes and tries to align data on these boundaries. The log format gives resilience in the face of power down and also gives good write performance, in some tests showing a two-fold improvement over ext4. There is a good description of the design of F2FS in the kernel documentation in Documentation/filesystems/f2fs.txt and there are references at the end of the chapter.

The `mkfs.f2fs` utility creates an empty F2FS filesystem with the label -1:

```
# mkfs.f2fs -l rootfs /dev/mmcblock0p1
# mount -t f2fs /dev/mmcblock0p1 /mnt
```

There isn't (yet) a tool to create F2FS filesystem images off-line.

FAT16/32

The old Microsoft filesystems, FAT16 and FAT32, continue to be important as a common format that is understood by most operating systems. When you buy an SD card or USB flash drive, it is almost certain to be formatted as FAT32 and, in some cases, the on-card microcontroller is optimized for FAT32 access patterns. Also, some boot ROMs require a FAT partition for the second stage bootloader, the TI OMAP-based chips for example. However, FAT formats are definitely not suitable for storing critical files because they are prone to corruption and make poor use of the storage space.

Linux supports FAT16 through the `msdos` filesystem and both FAT32 and FAT16 through the `vfat` filesystem. In most cases, you need to include the `vfat` driver. Then, to mount a device, say an SD card on the second `mmc` hardware adapter, you would type this:

```
# mount -t vfat /dev/mmcblock1p1 /mnt
```

In the past, there have been licensing issues with the `vfat` driver which may (or may not) infringe a patent held by Microsoft.

FAT32 has a limitation on the device size of 32 GiB. Devices of a larger capacity may be formatted using the Microsoft exFAT format and it is a requirement for SDXC cards. There is no kernel driver for exFAT, but it can be supported by means of a user-space FUSE driver. Since exFAT is proprietary to Microsoft there are certain to be licensing implications if you support this format on your device.

Read-only compressed filesystems

Compressing data is useful if you don't have quite enough storage to fit everything in. Both JFFS2 and UBIFS do on-the-fly data compression by default. However, if the files are never going to be written, as is usually the case with the root filesystem, you can achieve better compression ratios by using a read-only compressed filesystem. Linux supports several of these: `romfs`, `cramfs`, and `squashfs`. The first two are obsolete now, so I will describe only `squashfs`.

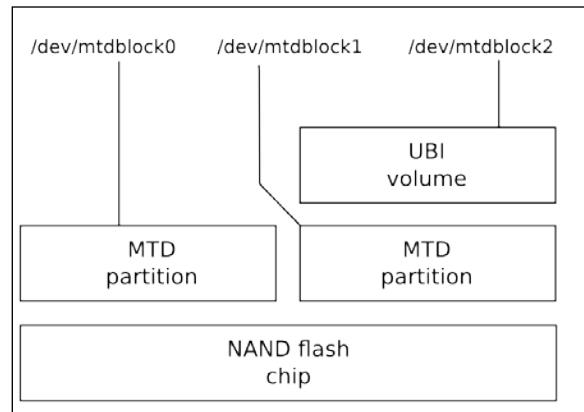
squashfs

`squashfs` was written by Phillip Louher in 2002 as a replacement for `cramfs`. It existed as a kernel patch for a long time, eventually being merged into mainline Linux in version 2.6.29 in 2009. It is very easy to use: you create a filesystem image using `mksquashfs` and install it to the flash memory:

```
$ mksquashfs rootfs rootfs.squashfs
```

The resulting filesystem is read-only so there is no mechanism to modify any of the files at runtime. The only way to update a `squashfs` filesystem is to erase the whole partition and program in a new image.

`squashfs` is not bad block-aware and so must be used with reliable flash memory such as NOR flash. It can be used on NAND flash as long as you use UBI to create an emulated, reliable, MTD volume on top of UBI. You have to enable kernel configuration `CONFIG_MTD_UBI_BLOCK`, which will create a read-only MTD block device for each UBI volume. The following diagram shows two MTD partitions, each with accompanying `mtdblock` devices. The second partition is also used to create a UBI volume which is exposed as a third, reliable, `mtdblock` device, which you can use for any read-only filesystem that is not bad block-aware:



Temporary filesystems

There are always some files that have a short lifetime or which have no significance after a reboot. Many such files are put into `/tmp`, and so it makes sense to keep these files from reaching permanent storage.

The temporary filesystem, `tmpfs`, is ideal for this purpose. You can create a temporary RAM-based filesystem by simply mounting `tmpfs`:

```
# mount -t tmpfs tmp_files /tmp
```

As with `procfs` and `sysfs`, there is no device node associated with `tmpfs` so you have to supply a place-holder string, `tmp_files` in the preceding example.

The amount of memory used will grow and shrink as files are created and deleted. The default maximum size is half the physical RAM. In most cases, it would be a disaster if `tmpfs` grew that large, so it is a very good idea to cap it with a `-o size` parameter. The parameter can be given in bytes, KiB (`k`), MiB (`m`), or GiB (`g`), for example:

```
mount -t tmpfs -o size=1m tmp_files /tmp
```

In addition to `/tmp`, some subdirectories of `/var` contain volatile data and it is good practice to use `tmpfs` for them as well, either by creating a separate filesystem for each or, more economically, by using symbolic links. Buildroot does it this way:

```
/var/cache -> /tmp  
/var/lock -> /tmp  
/var/log -> /tmp  
/var/run -> /tmp  
/var/spool -> /tmp  
/var/tmp -> /tmp
```

In the Yocto Project, `/run` and `/var/volatile` are `tmpfs` mounts with symbolic links pointing to them as shown here:

```
/tmp -> /var/tmp  
/var/lock -> /run/lock  
/var/log -> /var/volatile/log  
/var/run -> /run  
/var/tmp -> /var/volatile/tmp
```

Making the root filesystem read-only

You need to make your target device able to survive unexpected events including file corruption, and still be able to boot and achieve at least a minimum level of function. Making the root filesystem read-only is a key part of achieving this ambition because it eliminates accidental over-writes. Making it read-only is easy: replace `rw` with `ro` on the kernel command line or use an inherently read-only filesystem such as `squashfs`. However, you will find that there are a few files and directories that are traditionally writable:

- `/etc/resolv.conf`: This file is written by network configuration scripts to record the addresses of DNS name servers. The information is volatile, so you simply have to make it a symlink to a temporary directory, for example, `/etc/resolv.conf -> /var/run/resolv.conf`.
- `/etc/passwd`: This file, along with `/etc/group`, `/etc/shadow`, and `/etc/gshadow`, stores user and group names and passwords. They need to be symbolically linked to an area of persistent storage.
- `/var/lib`: Many applications expect to be able to write to this directory and to keep permanent data here as well. One solution is to copy a base set of files to a `tmpfs` filesystem at boot time and then bind mount `/var/lib` to the new location by putting a sequence of commands such as these into one of the boot scripts:

```
mkdir -p /var/volatile/lib
cp -a /var/lib/* /var/volatile/lib
mount --bind /var/volatile/lib /var/lib
```

- `/var/log`: This is the place where syslog and other daemons keep their logs. Generally, logging to flash memory is not desirable because of the many small write cycles it generates. A simple solution is to mount `/var/log` using `tmpfs`, making all log messages volatile. In the case of `syslogd`, BusyBox has a version that can log to a circular ring buffer.

If you are using the Yocto Project, you can create a read-only root filesystem by adding `IMAGE_FEATURES = "read-only-rootfs"` to `conf/local.conf` or to your image recipe.

Filesystem choices

So far we have looked at the technology behind solid state memory and at the many types of filesystems. Now it is time to summarize the options.

In most cases, you will be able to divide your storage requirements into these three categories:

- **Permanent, readable, and writable data**: Runtime configuration, network parameters, passwords, data logs, and user data
- **Permanent, read-only data**: Programs, libraries, and configurations files that are constant, for example, the root filesystem
- **Volatile data**: Temporary storage for example `/tmp`

The choices for read-write storage are as follows:

- **NOR**: UBIFS or JFFS2
- **NAND**: UBIFS, JFFS2, or YAFFS2
- **eMMC**: ext4 or F2FS



For read-only storage, you can use all of the above mounted with the `ro` attribute. Additionally, if you want to save space, you could use `squashfs`, in the case of NAND flash using UBI `mtdblock` device emulation to handle the bad blocks for you.

Finally, for volatile storage, there is only one choice, `tmpfs`.

Updating in the field

There have been several well-publicized security flaws, including Heartbleed (a bug in the OpenSSL libraries) and Shellshock (a bug in the bash shell), both of which could have serious consequences for embedded Linux devices that are currently deployed. For this reason alone, it is highly desirable to have a mechanism to update devices in the field so that you can fix security problems as they arise. There are other good reasons as well: to deploy other bug fixes and feature updates.

The guiding principle of update mechanisms is that they should do no harm, remembering Murphy's Law: if it can go wrong, it will go wrong, eventually. Any update mechanism must be:

- **Robust:** It must not render the device inoperable. I will talk about updates being atomic; either the system is updated successfully or not updated at all and continues to run as before.
- **Failsafe:** It must handle interrupted updates gracefully.
- **Secure:** It must not allow unauthorized updates, otherwise it will become an attack mechanism.

Atomicity can be achieved by having duplicates of the things you want to update and switching to the new copy when safe to do so.

Failsafety requires there to be a mechanism to detect a failed update, such as a hardware watchdog, and a known good copy of software to fall back on.

Security can be achieved in the case of updates that are local and attended through authentication by a password or PIN code. But, if the update is remote and automatic, some level of authentication via the network is needed. Ultimately, you may want to add a secure bootloader and signed update binaries.

Some components are easier to update than others. The bootloader is very difficult to update since there are usually hardware constraints that mean there can only be one bootloader, and so there cannot be a backup if the update fails. On the other hand, bootloaders are not often a cause of runtime bugs. The best advice is to avoid bootloader updates in the field.

Granularity: file, package, or image?

This is the big question, and depends on your overall system design and your desired level of robustness.

File updates can be made atomic: the technique is to write the new content to a temporary file in the same filesystem and then use the POSIX `rename(2)` function to move it over the old file. It works because `rename` is guaranteed to be atomic. However, this is only one part of the problem because there will be dependencies between files which need to be considered.

Updating at the level of packages (`RPM`, `dpkg`, or `ipk`) is a better option, assuming that you have a runtime package manager. This, after all, is how desktop distributions have been doing it for years. The package manager has a database of updates and can keep track of those which have been updated and those that haven't. Each package has an update script that is designed to make sure that the package update is atomic. The great advantage is that you can update existing packages, install new ones, and delete obsolete ones with ease. If you are using a root filesystem that is mounted as read-only, you will have to temporarily remount read-write while updating, which opens up a small window for corruption.

Package managers do have downsides as well. They are not able to update kernel or other images in raw flash memory. After devices have been deployed and updated several times, you may end up with a large number of combinations of packages and package versions, which will complicate QA for each new update cycle. Package managers are not bulletproof in the event of power failure during an update.

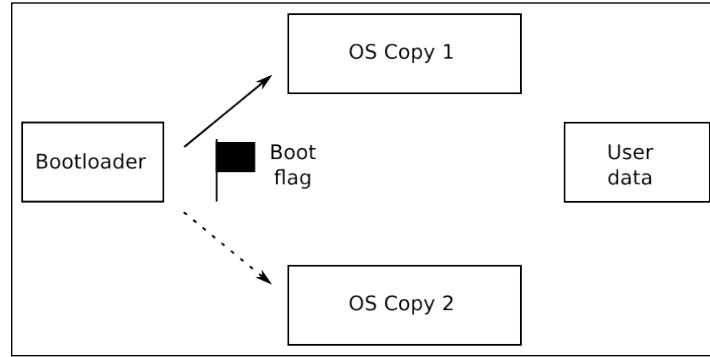
The third option is to update whole system images: the kernel, the root filesystem, user applications, and so on.

Atomic image update

In order to make the update atomic, we need two things: a second copy of the operating system that can be used during the update, and a mechanism in the bootloader to select which copy of the operating system to load. The second copy may be exactly the same as the first, resulting in full redundancy of the operating system, or it may be a small operating system dedicated to updating the main one.

Creating a Storage Strategy

In the first scheme ,there are two copies of the operating system, each comprised of the Linux kernel, the root filesystem, and system applications, as shown in the following diagram:

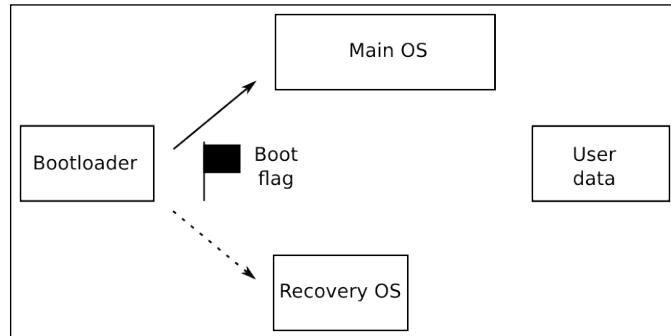


Initially, the boot flag is not set, so the bootloader loads copy 1. To install an update, the updater application, which is part of the operating system, overwrites copy 2. When complete, it sets the boot flag and reboots. Now, the bootloader will load the new operating system. When a further update is installed, the updater in copy 2 overwrites copy 1 and clears the boot flag and so you ping-pong between the two copies.

If an update fails, the boot flag is not changed and the last good operating system is used. Even if the update consists of several components, a kernel image, a DTB, a root filesystem, and a system application filesystem, the whole update is atomic because the boot flag is only updated when all updates are completed.

The main drawback with this scheme is that it requires storage for two copies of the operating system.

You can reduce storage requirements by keeping a minimal operating system purely for updating the main one, as shown in the following diagram:



When you want to install an update, set the boot flag and reboot. Once the recovery operating system is running, it starts the updater which overwrites the main operating system images. When done, it clears the boot flag and reboots again, this time loading the new main operating system.

The recovery operating system is usually a lot smaller than the main operating system, maybe only a few megabytes, and so the storage overhead is not great. In fact, this is the scheme adopted by Android. The main operating system is several hundred megabytes, but the recovery mode operating system is a simple ramdisk of a few megabytes only.

Further reading

The following resources have further information about the topics introduced in this chapter:

- *XIP: The past, the present... the future?*, Vitaly Wool, presentation at FOSDEM 2007: https://archive.fosdem.org/2007/slides/devrooms/embedded/Vitaly_Wool_XIP.pdf
- *General MTD documentation*, <http://www.linux-mtd.infradead.org/doc/general.html>
- *Optimizing Linux with cheap flash drives*, Arnd Bergmann: <http://lwn.net/Articles/428584/>
- *Flash memory card design*: <https://wiki.linaro.org/WorkingGroups/KernelArchived/Projects/FlashCardSurvey>
- *eMMC/SSD File System Tuning Methodology*: http://elinux.org/images/b/b6/EMMC-SSD_File_System_Tuning_Methodology_v1.0.pdf
- *Flash-Friendly File System (F2FS)*: http://elinux.org/images/1/12/EIc2013_Hwang.pdf
- *An f2fs teardown*: <http://lwn.net/Articles/518988/>
- *Building Murphy-compatible embedded Linux systems*, Gilad Ben-Yossef: <https://www.kernel.org/doc/ols/2005/ols2005v1-pages-21-36.pdf>

Summary

Flash memory has been the storage technology of choice for embedded Linux from the beginning and over the years Linux has gained very good support, from low-level drivers up to flash-aware filesystems, the latest being UBIFS.

However, as the rate at which new flash technologies are introduced increases, it is becoming harder to keep pace with the changes at the high end. System designers are increasingly turning to managed flash in the form of eMMC to provide a stable hardware and software interface which is independent of the memory chips inside. Embedded Linux developers are beginning to get to grips with these new chips. Support for TRIM in ext4 and F2FS is well established, and it is slowly finding its way into the chips themselves. Also, the appearance of new filesystems that are optimized to manage flash, such as F2FS, is a welcome step forward.

However, the fact remains that flash memory is not the same as a hard disk drive. You have to be careful to minimize the number of filesystem writes – especially as the higher density TLC chips may be able to support as few as 1,000 erase cycles.

Finally, it is essential to have a strategy for updating the files and images stored on the device in the field. A crucial part of this is the decision to use a package manager or not. A package manager gives you flexibility, but cannot give you a fully Murphy proof update solution. Your choice depends on the balance between convenience and robustness.

The next chapter describes how you control the hardware components of your system through the use of device drivers, both in the conventional sense of drivers that are part of the kernel, and also the extent to which you can control hardware from the user space.

8

Introducing Device Drivers

Kernel device drivers are the mechanism through which the underlying hardware is exposed to the rest of the system. As a developer of embedded systems, you need to know how device drivers fit into the overall architecture and how to access them from user space programs. Your system will probably have some novel pieces of hardware and you will have to work out a way of accessing them. In many cases, you will find that there are device drivers provided for you and you can achieve everything you want without writing any kernel code. For example, you can manipulate GPIO pins and LEDs using files in `sysfs`, and there are libraries to access serial buses, including SPI and I2C.

There are many places to find out how to write a device driver, but few to tell you why you would want to and the choices you have in doing so. That is what I want to cover here. However, remember that this is not a book dedicated to writing kernel device drivers and that the information given here is to help you navigate the territory but not necessarily to set up home there. There are many good books and articles that will help you to write device drivers, some of which are listed at the end of this chapter.

The role of device drivers

As mentioned in *Chapter 4, Porting and Configuring the Kernel*, one of the functions of the kernel is to encapsulate the many hardware interfaces of a computer system and present them in a consistent manner to user-space programs. There are frameworks designed to make it easy to write the interface logic for a device in the kernel and you can integrate it with the kernel: this is a device driver, the piece of code that mediates between the kernel above it and the hardware below. A device driver is a piece of software that controls physical devices such as a UART or an MMC controller, or virtual devices such as the null device (`/dev/null`) or a ramdisk. One driver may control multiple devices of the same kind.

Kernel device driver code runs at a high privilege level, as does the rest of the kernel. It has full access to the processor address space and hardware registers. It can handle interrupts and DMA transfers. It can make use of the sophisticated kernel infrastructure for synchronization and memory management. There is a downside to this, which is that if something goes wrong in a buggy driver, it can go really wrong and bring the system down. Consequently, there is a principle that device drivers should be as simple as possible, just providing information to applications where the real decisions are made. You often hear this being expressed as *no policy in the kernel*.

In Linux, there are three main types of device driver:

- **character**: This is for unbuffered I/O with a rich range of functions and a thin layer between the application code and the driver. It is the first choice when implementing custom device drivers.
- **block**: This has an interface tailored for block I/O to and from mass storage devices. There is a thick layer of buffering designed to make disk reads and writes as fast as possible, which makes it unsuitable for anything else.
- **network**: This is similar to a block device but is used for transmitting and receiving network packets rather than disk blocks.

There is also a fourth type that presents itself as a group of files in one of the pseudo filesystems. For example, you might access the GPIO driver through a group of files in `/sys/class/gpio`, as I will describe later on in this chapter. Let's begin by looking in more detail at the three basic device types.

Character devices

These devices are identified in user space by a filename: if you want to read from a UART, you open the device node, for example, the first serial port on the ARM Versatile Express would be `/dev/ttAMA0`. The driver is identified differently in the kernel, using the major number which, in the example given, is 204. Since the UART driver can handle more than one UART, there is a second number, called the minor number, which identifies a specific interface, 64, in this case:

```
# ls -l /dev/ttAMA*
crw-rw---- 1 root      root      204,   64 Jan  1  1970 /dev/ttAMA0
crw-rw---- 1 root      root      204,   65 Jan  1  1970 /dev/ttAMA1
crw-rw---- 1 root      root      204,   66 Jan  1  1970 /dev/ttAMA2
crw-rw---- 1 root      root      204,   67 Jan  1  1970 /dev/ttAMA3
```

The list of standard major and minor numbers can be found in the kernel documentation, in `Documentation/devices.txt`. The list does not get updated very often and does not include the `ttyAMA` device described in the preceding paragraph. Nevertheless, if you look at the source code in `drivers/tty/serial/amba-p1011.c`, you will see that the major and minor numbers are declared:

```
#define SERIAL_AMBA_MAJOR      204  
#define SERIAL_AMBA_MINOR      64
```

Where there is more than one instance of a device, the naming convention for the device nodes is `<base name><interface number>`, for example, `ttyAMA0`, `ttyAMA1`, and so on.

As I mentioned in *Chapter 5, Building a Root Filesystem*, the device nodes can be created in several ways:

- `devtmpfs`: The node that is created when the device driver registers a new device interface using a base name supplied by the driver (`ttyAMA`) and an instance number.
- `udev` or `mdev` (without `devtmpfs`): Essentially the same as with `devtmpfs`, except that a user-space daemon program has to extract the device name from `sysfs` and create the node. I will talk about `sysfs` later.
- `mknod`: If you are using static device nodes, they are created manually using `mknod`.

You may have the impression from the numbers I have used above that both major and minor numbers are 8-bit numbers in the range 0 to 255. In fact, from Linux 2.6 onwards, the major number is 12 bits long, which gives valid numbers from 1 to 4,095, and the minor number is 20 bits, from 0 to 1,048,575.

When you open a device node, the kernel checks to see whether the major and minor numbers fall into a range registered by a device driver of that type (a character or block). If so, it passes the call to the driver, otherwise the open call fails. The device driver can extract the minor number to find out which hardware interface to use. If the minor number is out of range, it returns an error.

To write a program that accesses a device driver, you have to have some knowledge of how it works. In other words, a device driver is not the same as a file: the things you do with it change the state of the device. A simple example is the pseudo random number generator, `urandom`, which returns bytes of random data every time you read it. Here is a program that does just that:

```
#include <stdio.h>  
#include <sys/types.h>
```

```
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(void)
{
    int f;
    unsigned int rnd;
    int n;
    f = open("/dev/urandom", O_RDONLY);
    if (f < 0) {
        perror("Failed to open urandom");
        return 1;
    }
    n = read(f, &rnd, sizeof(rnd));
    if (n != sizeof(rnd)) {
        perror("Problem reading urandom");
        return 1;
    }
    printf("Random number = 0x%x\n", rnd);
    close(f);
    return 0;
}
```

The nice thing about the Unix driver model is that, once we know that there is a device named `urandom` and that every time we read from it, it returns a fresh set of pseudo random data, we don't need to know anything else about it. We can just use normal functions such as `open(2)`, `read(2)`, and `close(2)`.

We could use the stream I/O functions `fopen(3)`, `fread(3)`, and `fclose(3)` instead, but the buffering implicit in these functions often causes unexpected behavior. For example, `fwrite(3)` usually only writes to the user-space buffer, not to the device. We would need to call `fflush(3)` to force the buffer to be written out.



Don't use stream I/O functions such as `fread(3)` and `fwrite(3)` when calling device drivers.

Block devices

Block devices are also associated with a device node, which also has major and minor numbers.



Although character and block devices are identified using major and minor numbers, they are in different namespaces. A character driver with a major number 4 is in no way related to a block driver with a major number 4.

With block devices, the major number is used to identify the device driver and the minor number is used to identify the partition. Let's look at the MMC driver as an example:

```
# ls -l /dev/mmcblk*
brw----- 1 root root 179,    0 Jan  1  1970 /dev/mmcblk0
brw----- 1 root root 179,    1 Jan  1  1970 /dev/mmcblk0p1
brw----- 1 root root 179,    2 Jan  1  1970 /dev/mmcblk0p2
brw----- 1 root root 179,    8 Jan  1  1970 /dev/mmcblk1
brw----- 1 root root 179,    9 Jan  1  1970 /dev/mmcblk1p1
brw----- 1 root root 179,   10 Jan  1  1970 /dev/mmcblk1p2
```

The major number is 179 (look it up in `devices.txt!`). The minor numbers are used in ranges to identify different `mmc` devices and the partitions of the storage medium on that device. In the case of the `mmcblk` driver, the ranges are eight minor numbers per device: the minor numbers from 0 to 7 are for the first device, the numbers from 8 to 15 are for the second, and so on. Within each range, the first minor number represents the entire device as raw sectors, and the others represent up to seven partitions.

You are probably aware of the SCSI disk driver, known as `sd`, which is used to control a range of disks that use the SCSI command set, which includes SCSI, SATA, USB mass storage, and **UFS (universal flash storage)**. It has the major number eight and ranges of 16 minor numbers per interface (or disk). The minor numbers from 0 to 15 are for the first interface, with device nodes named `sda` up to `sda15`, the numbers from 16 to 31 are for the second disk with device nodes `sdb` up to `sdb15`, and so on. This continues up to the sixteenth disk from 240 to 255, with the node name `sdp`. There are other major numbers reserved for them because SCSI disks are so popular, but we needn't worry about that here.

The partitions are created using utilities such as `fdisk`, `sfidisk`, or `parted`. An exception is raw flash memory: the partition information for the MTD driver is part of the kernel command line or in the device tree, or one of the other methods described in *Chapter 7, Creating a Storage Strategy*.

A user-space program can open and interact directly with a block device through the device node. This is not a common thing to do, and is usually for performing administrative operations such as partitioning, formatting with a filesystem, and mounting. Once the filesystem is mounted, you interact with the block device indirectly through the files in that filesystem.

Network devices

Network devices are not accessed through device nodes and they do not have major and minor numbers. Instead, a network device is allocated a name by the kernel, based on a string and an instance number. Here is an example of the way a network driver registers an interface:

```
my_netdev = alloc_netdev(0, "net%d", NET_NAME_UNKNOWN,
    netdev_setup);
ret = register_netdev(my_netdev);
```

This creates a network device named `net0` the first time it is called, `net1` the second, and so on. More common names are `lo`, `eth0`, and `wlan0`.

Note that this is the name it starts off with; device managers, such as `udev`, may change to something different later on.

Usually, the network interface name is only used when configuring the network using utilities such as `ip` and `ifconfig` to establish a network address and route. Thereafter, you interact with the network driver indirectly by opening sockets, and let the network layer decide how to route them to the right interface.

However, it is possible to access network devices directly from user space by creating a socket and using the `ioctl` commands listed in `include/linux/sockios.h`. For example, this program uses `SIOCGIFHWADDR` to query the driver for the hardware (MAC) address:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <linux/sockios.h>
#include <net/if.h>
int main (int argc, char *argv[])
{
    int s;
    int ret;
    struct ifreq ifr;
    int i;
```

```
if (argc != 2) {
    printf("Usage %s [network interface]\n", argv[0]);
    return 1;
}
s = socket(PF_INET, SOCK_DGRAM, 0);
if (s < 0) {
    perror("socket");
    return 1;
}
strcpy(ifr.ifr_name, argv[1]);
ret = ioctl(s, SIOCGIFHWADDR, &ifr);
if (ret < 0) {
    perror("ioctl");
    return 1;
}
for (i = 0; i < 6; i++)
    printf("%02x:", (unsigned char)ifr.ifr_hwaddr.sa_data[i]);
printf("\n");
close(s);
return 0;
}
```

This is a standard device, `ioctl`, which is handled by the network layer on the driver's behalf, but it is possible to define your own `ioctl` numbers and handle them in a custom network driver.

Finding out about drivers at runtime

Once you have a running Linux system, it is useful to know which device drivers are loaded and what state they are in. You can find out a lot by reading the files in `/proc` and `/sys`.

First of all, you can list the character and block device drivers currently loaded and active by reading `/proc/devices`:

```
# cat /proc/devices
```

```
Character devices:
```

```
1 mem
2 pty
3 ttyp
4 /dev/vc/0
4 tty
```

```
4 ttys
5 /dev/tty
5 /dev/console
5 /dev/ptmx
7 vcs
10 misc
13 input
29 fb
81 video4linux
89 i2c
90 mtd
116 alsa
128 ptm
136 pts
153 spi
180 usb
189 usb_device
204 ttySC
204 ttyAMA
207 ttymxc
226 drm
239 ttyLP
240 ttyTHS
241 ttySiRF
242 ttyPS
243 ttyWMT
244 ttyAS
245 ttyO
246 ttyMSM
247 ttyAML
248 bsg
249 iio
250 watchdog
251 ptp
252 pps
253 media
254 rtc
```

Block devices:

```
259 blkext
7 loop
8 sd
11 sr
```

```
31 mtdblock
65 sd
66 sd
67 sd
68 sd
69 sd
70 sd
71 sd
128 sd
129 sd
130 sd
131 sd
132 sd
133 sd
134 sd
135 sd
179 mmc
```

For each driver, you can see the major number and the base name. However, this does not tell you how many devices each driver is attached to. It only shows `ttyAMA` but gives you no clue that it is attached to four real UARTS. I will come back to that later when I look at `sysfs`. If you are using a device manager such as `mdev`, `udev`, or `devtmpfs`, you can list the character and block device interfaces by looking in `/dev`.

You can also list network interfaces using `ifconfig` or `ip`:

```
# ip link show

1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state
UNKNOWN mode DEFAULT
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

2: eth0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc
pfifo_fast state DOWN mode DEFAULT qlen 1000
    link/ether 54:4a:16:bb:b7:03 brd ff:ff:ff:ff:ff:ff

3: usb0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
pfifo_fast state UP mode DEFAULT qlen 1000
    link/ether aa:fb:7f:5e:a8:d5 brd ff:ff:ff:ff:ff:ff
```

You can also find out about devices attached to USB or PCI buses using the well-known commands `lsusb` and `lspci`. There is information about them in the respective manuals and plenty of online guides, so I will not describe them any further here.

The really interesting information is in `sysfs`, which is the next topic.

Getting information from sysfs

You can define `sysfs` in a pedantic way as a representation of kernel objects, attributes and relationships. A kernel object is a directory, an attribute is a file, and a relationship is a symbolic link from one object to another.

From a more practical point of view, since the Linux device driver model, which was introduced in version 2.6, represents all devices and drivers as kernel objects. You can see the kernel's view of the system laid out before you by looking in `/sys`, as shown here:

```
# ls /sys
block  bus  class  dev  devices  firmware  fs  kernel  module
power
```

In the context of discovering information about devices and drivers, I will look at three of the directories: `devices`, `class`, and `block`.

The devices: `/sys/devices`

This is the kernel's view of the devices discovered since boot and how they are connected to each other. It is organized at the top level by the system bus, so what you see varies from one system to another. This is the QEMU emulation of the Versatile Express:

```
# ls
/sys/devices
armv7_cortex_a9  platform      system
breakpoint        software       virtual
```

There are three directories that are present on all systems:

- `system`: This contains devices at the heart of the system, including CPUs and clocks.
- `virtual`: This contains devices that are memory-based. You will find the memory devices that appear as `/dev/null`, `/dev/random`, and `/dev/zero` in `virtual/mem`. You will find the loopback device, `lo`, in `virtual/net`.
- `platform`: This is a catch-all for devices that are not connected via a conventional hardware bus. This may be almost everything on an embedded device.

The other devices appear in directories that correspond to actual system buses. For example, the PCI root bus, if there is one, appears as `pci0000:00`.

Navigating this hierarchy is quite hard because it requires some knowledge of the topology of your system and the pathnames become quite long and hard to remember. To make life easier, `/sys/class` and `/sys/block` offer two different views of the devices.

The drivers: `/sys/class`

This is a view of the device drivers presented by their type, in other words, it is a software view rather than a hardware view. Each of the subdirectories represents a class of driver and is implemented by a component of the driver framework. For example, UART devices are managed by the `tty` layer and you will find them in `/sys/class/tty`. Likewise, you will find network devices in `/sys/class/net`, input devices such as the keyboard, the touchscreen, and the mouse in `/sys/class/input`, and so on.

There is a symbolic link in each subdirectory for each instance of that type of device pointing to its representation in `/sys/device`.

To take a concrete example, let's look at `/sys/class/tty/ttyAMA0`:

```
# cd /sys/class/tty/ttyAMA0/  
# ls  
close_delay      flags          line          uartclk  
closing_wait     io_type        port          uevent  
custom_divisor   iomem_base    power         xmit_fifo_size  
dev              iomem_reg_shift subsystem  
device           irq            type
```

The link `device` references the hardware node for the device and `subsystem` points back to `/sys/class/tty`. The others are attributes of the device. Some are specific to a UART, such as `xmit_fifo_size` and others apply to many types of device such as the interrupt number, `irq`, and the device number `dev`. Some attribute files are writable and allow you to tune parameters in the driver at runtime.

The `dev` attribute is particularly interesting. If you look at its value, you will find the following:

```
# cat /sys/class/tty/ttyAMA0/dev  
204:64
```

These are the major and minor numbers of this device. This attribute is created when the driver registered this interface and it is from this file that `udev` and `mdev` read that information if they are being used without the help of `devtmpfs`.

The block drivers: /sys/block

There is one more view of the device model that is important: the block driver view that you will find in `/sys/block`. There is a subdirectory for each block device. This example is taken from a BeagleBone Black:

```
# ls /sys/block/  
  
loop0  loop4  mmcblk0      ram0   ram12  ram2  ram6  
loop1  loop5  mmcblk1      ram1   ram13  ram3  ram7  
loop2  loop6  mmcblk1boot0 ram10  ram14  ram4  ram8  
loop3  loop7  mmcblk1boot1 ram11  ram15  ram5  ram9
```

If you look into `mmcblk1`, which is the eMMC chip on this board, you can see the attributes of the interface and the partitions within it:

```
# cd /sys/block/mmcblk1  
# ls  
  
alignment_offset  ext_range      mmcblk1p1  ro  
bdi              force_ro       mmcblk1p2  size  
capability       holders        power     slaves  
dev               inflight       queue    stat  
device            mmcblk1boot0  range    subsystem  
discard_alignment mmcblk1boot1 removable uevent
```

The conclusion, then, is that you can learn a lot about the devices (the hardware) and the drivers (the software) that are present on a system by reading `sysfs`.

Finding the right device driver

A typical embedded board is based on a reference design from the manufacturer with changes to make it suitable for a particular application. It may have a temperature sensor attached via I2C, lights and buttons connected via GPIO pins, an external Ethernet MAC, a display panel via a MIPI interface, or many other things. Your job is to create a custom kernel to control all of that, so where do you start?

Some things are so simple that you can write user space code to handle them. GPIOs and simple peripherals connected via I2C or SPI are easy to control from user space, as I will explain later.

Other things need a kernel driver so you need to know how to find one and incorporate it into your build. There is no simple answer, but here are some places to look.

The most obvious place to look is the driver support page on the manufacturer's website, or you could ask them directly. In my experience, this seldom gets the result you want; hardware manufacturers are not particularly Linux-savvy and they often give you misleading information. They may have proprietary drivers as binary blobs or they may have source code, but for a different version of the kernel than the one you have. So, by all means try this route. I will always try to find an open source driver for the task in hand.

There may be support in your kernel already: there are many thousands of drivers in mainline Linux and there are many vendor-specific drivers in the vendor kernels. Begin by running `make menuconfig` (or `xconfig`) and search for the product name or number. If you do not find an exact match, try more generic searches, allowing for the fact that most drivers handle a range of products from the same family. Next, try searching through the code in the drivers directory (`grep` is your friend here). Always make sure you are running the latest kernel for your board: later kernels generally have more device drivers.

If you still don't have a driver, you can try searching online and asking in the relevant forums to see if there is a driver for a different version of Linux. If you find one, you will have to backport it to your kernel. If the kernel versions are similar, it may be easy, but if they are more than 12 to 18 months apart, the chances are that the interfaces will have changed to the extent that you will have to rewrite a chunk of the driver to integrate it with your kernel. You may want to outsource this work. If all of the above fails, you will have to find a solution yourself.

Device drivers in user-space

Before you start writing a device driver, pause for a moment to consider whether it is really necessary. There are generic device drivers for many common types of device that allow you to interact with hardware directly from user space without having to write a line of kernel code. User space code is certainly easier to write and debug. It is also not covered by the GPL, although I don't feel that is a good reason in itself to do it this way.

They fall into two broad categories: those that you control through files in `sysfs`, including GPIO and LEDs, and serial buses that expose a generic interface through a device node, such as I2C.

GPIO

General Purpose Input/Output (GPIO) is the simplest form of digital interface since it gives you direct access to individual hardware pins, each of which can be configured as input or output. GPIO can even be used to create higher level interfaces such as I2C or SPI by manipulating each bit in the software, a technique that is called bit banging. The main limitation is the speed and accuracy of the software loops and the number of CPU cycles you want to dedicate to them. Generally speaking, it is hard to achieve timer accuracy better than a millisecond with kernels compiled with `CONFIG_PREEMPT`, and 100 microseconds with `RT_PREEMPT`, as we shall see in *Chapter 14, Real-time Programming*. More common use cases for GPIO are for reading push buttons and digital sensors and controlling LEDs, motors, and relays.

Most SoCs have a lot of GPIO bits which are grouped together in GPIO registers, usually 32 bits per register. On-chip GPIO bits are routed through to GPIO pins on the chip package via a multiplexer, known as a pin mux, which I will describe later. There may be additional GPIO bits available off-chip in the power management chip, and in dedicated GPIO extenders, connected through I2C or SPI buses. All this diversity is handled by a kernel subsystem known as `gpiolib`, which is not actually a library but the infrastructure GPIO drivers use to expose IO in a consistent way.

There are details about the implementation of `gpiolib` in the kernel source in `Documentation/gpio` and the drivers themselves are in `drivers/gpio`.

Applications can interact with `gpiolib` through files in the `/sys/class/gpio` directory. Here is an example of what you will see in there on a typical embedded board (a BeagleBone Black):

```
# ls /sys/class/gpio  
export gpiochip0  gpiochip32  gpiochip64  gpiochip96  unexport
```

The `gpiochip0` to `gpiochip96` directories represent four GPIO registers, each with 32 GPIO bits. If you look in one of the `gpiochip` directories, you will see the following:

```
# ls /sys/class/gpio/gpiochip96/  
base  label  ngpio  power  subsystem  uevent
```

The file `base` contains the number of the first GPIO pin in the register and `ngpio` contains the number of bits in the register. In this case, `gpiochip96/base` is 96 and `gpiochip96/ngpio` is 32, which tells you that it contains GPIO bits 96 to 127. It is possible for there to be a gap between the last GPIO in one register and the first GPIO in the next.

To control a GPIO bit from user space, you first have to export it from kernel space, which you do by writing the GPIO number to `/sys/class/gpio/export`. This example shows the process for GPIO 48:

```
# echo 48 > /sys/class/gpio/export
# ls /sys/class/gpio
export      gpio48      gpiochip0    gpiochip32  gpiochip64
gpiochip96  unexport
```

Now there is a new directory, `gpio48`, which contains the files you need to control the pin. Note that if the GPIO bit is already claimed by the kernel, you will not be able to export it in this way.

The directory `gpio48` contains these files:

```
# ls /sys/class/gpio/gpio48
active_low  direction  edge  power  subsystem  uevent  value
```

The pin begins as an input. To change it to an output, write out to the `direction` file. The file `value` contains the current state of the pin, which is 0 for low and 1 for high. If it is an output, you can change the state by writing 0 or 1 to `value`. Sometimes, the meaning of low and high is reversed in hardware (hardware engineers enjoy doing that sort of thing), so writing 1 to `active_low` inverts the meaning so that a low voltage is reported as 1 in `value` and a high voltage as 0.

You can remove a GPIO from user space control by writing the GPIO number to `/sys/class/gpio/unexport`.

Handling interrupts from GPIO

In many cases, a GPIO input can be configured to generate an interrupt when it changes state, which allows you to wait for the interrupt rather than polling in an inefficient software loop. If the GPIO bit can generate interrupts, the file `edge` exists. Initially, it has the value `none`, meaning that it does not generate interrupts. To enable interrupts, you can set it to one of these values:

- **rising**: Interrupt on rising edge
- **falling**: Interrupt on falling edge
- **both**: Interrupt on both rising and falling edges
- **none**: No interrupts (default)

You can wait for an interrupt using the `poll()` function with `POLLPRI` as the event. If you want to wait for a rising edge on GPIO 48, you first enable interrupts:

```
# echo 48 > /sys/class/gpio/export
# echo rising > /sys/class/gpio/gpio48/edge
```

Then, you use `poll()` to wait for the change, as shown in this code example:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <poll.h>

int main (int argc, char *argv[])
{
    int f;
    struct pollfd poll_fds [1];
    int ret;
    char value[4];
    int n;
    f = open ("/sys/class/gpio/gpio48/value", O_RDONLY);
    if (f == -1) {
        perror ("Can't open gpio48");
        return 1;
    }
    poll_fds[0].fd = f;
    poll_fds[0].events = POLLPRI | POLLERR;
    while (1) {
        printf ("Waiting\n");
        ret = poll(poll_fds, 1, -1);
        if (ret > 0) {
            n = read(f, &value, sizeof(value));
            printf ("Button pressed: read %d bytes, value=%c\n",
                   n, value[0]);
        }
    }
    return 0;
}
```

LEDs

LEDs are often controlled through a GPIO pin, but there is another kernel subsystem that offers more specialized control specific to the purpose. The `leds` kernel subsystem adds the ability to set brightness, should the LED have that ability, and can handle LEDs connected in other ways than a simple GPIO pin. It can be configured to trigger the LED on an event such as block device access or just a heartbeat to show that the device is working. There is more information in Documentation/leds/ and the drivers are in drivers/leds/.

As with GPIOs, LEDs are controlled through an interface in sysfs, in /sys/class/leds. The LEDs have names in the form devicename:colour:function, as shown here:

```
# ls /sys/class/leds
beaglebone:green:heartbeat  beaglebone:green:usr2
beaglebone:green:mmc0       beaglebone:green:usr3
```

This shows one individual LED:

```
# ls /sys/class/leds/beaglebone:green:usr2
brightness      max_brightness   subsystem      uevent
device          power           trigger
```

The `brightness` file controls the brightness of the LED and can be a number between 0 (off) and `max_brightness` (fully on). If the LED doesn't support intermediate brightness, any non-zero value turns it on and zero turns it off. The file `trigger` lists the events that trigger the LED to turn on. The list of triggers is implementation-dependent. Here is an example:

```
# cat /sys/class/leds/beaglebone:green:heartbeat/trigger
none mmc0 mmc1 timer oneshot [heartbeat] backlight gpio cpu0
default-on
```

The trigger currently selected is shown in square brackets. You can change it by writing one of the other triggers to the file. If you want to control the LED entirely through `brightness`, select `none`. If you set the trigger to `timer`, two extra files appear that allow you to set the on and off times in milliseconds:

```
# echo timer > /sys/class/leds/beaglebone:green:heartbeat/trigger
# ls /sys/class/leds/beaglebone:green:heartbeat
brightness  delay_on   max_brightness  subsystem  uevent
delay_off   device     power           trigger
# cat /sys/class/leds/beaglebone:green:heartbeat/delay_on
500
# cat /sys/class/leds/beaglebone:green:heartbeat/delay_off
500
#
```

If the LED has on-chip timer hardware, the blinking takes place without interrupting the CPU.

I2C

I2C is a simple low speed 2-wire bus that is common on embedded boards, typically used to access peripherals which are not on the SoC board such as display controllers, camera sensors, GPIO extenders, and the like. There is a related standard known as SMBus (system management bus) that is found on PCs, that is used to access temperature and voltage sensors. SMBus is a subset of I2C.

I2C is a master-slave protocol, with the master being one or more host controllers on the SoC. Slaves have a 7-bit address assigned by the manufacturer – read the data sheet – allowing up to 128 nodes per bus, but 16 are reserved, so only 112 nodes are allowed in practice. The bus speed is 100 KHz in standard mode, or up to 400 KHz in fast mode. The protocol allows read and write transactions between the master and slave of up to 32 bytes. Frequently, the first byte is used to specify a register on the peripheral and the remaining bytes are the data read from or written to that register.

There is one device node for each host controller, for example, this SoC has four:

```
# ls -l /dev/i2c*
crw-rw---- 1 root i2c 89, 0 Jan  1 00:18 /dev/i2c-0
crw-rw---- 1 root i2c 89, 1 Jan  1 00:18 /dev/i2c-1
crw-rw---- 1 root i2c 89, 2 Jan  1 00:18 /dev/i2c-2
crw-rw---- 1 root i2c 89, 3 Jan  1 00:18 /dev/i2c-3
```

The device interface provides a series of `ioctl` commands that query the host controller and send read and write commands to I2C slaves. There is a package named `i2c-tools` which uses this interface to provide basic command-line tools to interact with I2C devices. The tools are as follows:

- `i2cdetect`: This lists the I2C adapters and probes the bus
- `i2cdump`: This dumps data from all the registers of an I2C peripheral
- `i2cget`: This reads data from an I2C slave
- `i2cset`: This writes data to an I2C slave

The `i2c-tools` package is available in Buildroot and the Yocto Project, as well as most mainstream distributions. So long as you know the address and protocol of the slave, writing a user space program to talk to the device is straightforward:

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
```

```
#include <i2c-dev.h>
#include <sys/ioctl.h>
#define I2C_ADDRESS 0x5d
#define CHIP_REVISION_REG 0x10

void main (void)
{
    int f_i2c;
    int val;

    /* Open the adapter and set the address of the I2C device */
    f_i2c = open ("/dev/i2c-1", O_RDWR);
    ioctl (f_i2c, I2C_SLAVE, I2C_ADDRESS);

    /* Read 16-bits of data from a register */
    val = i2c_smbus_read_word_data(f, CHIP_REVISION_REG);
    printf ("Sensor chip revision %d\n", val);
    close (f_i2c);
}
```

Note that the header `i2c-dev.h` is the one from the `i2c-tools` package, not the one from the Linux kernel headers. The `i2c_smbus_read_word_data()` function is written inline in `i2c-dev.h`.

There is more information about the Linux implementation of I2C in `Documentation/i2c/dev-interface`. The host controller drivers are in `drivers/i2c/busses`.

SPI

The serial peripheral interface bus is similar to I2C, but is a lot faster, up to the low MHz. The interface uses four wires with separate send and receive lines which allows it to operate in full duplex. Each chip on the bus is selected with a dedicated chip select line. It is commonly used to connect to touchscreen sensors, display controllers, and serial NOR flash devices.

As with I2C, it is a master-slave protocol, with most SoCs implementing one or more master host controllers. There is a generic SPI device driver which you can enable through the kernel configuration `CONFIG_SPI_SPIDEV`. It creates a device node for each SPI controller which allows you to access SPI chips from user space. The device nodes are named `spidev [bus] . [chip select]`:

```
# ls -l /dev/spi*
crw-rw---- 1 root root 153, 0 Jan 1 00:29 /dev/spidev1.0
```

For examples of using the `spidev` interface, refer to the example code in `Documentation/spi`.

Writing a kernel device driver

Eventually, when you have exhausted all the user-space options above, you will find yourself having to write a device driver to access a piece of hardware attached to your device. While this is not the time or place to delve into details, it is worth considering the options. Character drivers are the most flexible and should cover 90% of all your needs; network devices apply if you are working with a network interface, and block devices are for mass storage. The task of writing a kernel driver is complex and beyond the scope of this book. There are some references at the end that will help you on your way. In this section, I want to outline the options available for interacting with a driver – a topic not normally covered – and show you the basic bones of a driver.

Designing a character device interface

The main character device interface is based on a stream of bytes, as you would have with a serial port. However, many devices don't fit this description: a controller for a robot arm needs functions to move and rotate each joint, for example. Luckily, there are other ways to communicate with device drivers that just `read(2)` and `write(2)`:

- `ioctl`: The `ioctl` function allows you to pass two arguments to your driver which can have any meaning you like. By convention, the first argument is a command which selects one of several functions in your driver, and the second is a pointer to a structure which serves as a container for the input and output parameters. This is a blank canvas that allows you to design any program interface you like and it is pretty common when the driver and application are closely linked and written by the same team. However, `ioctl` is deprecated in the kernel and you will find it hard to get any drivers with new uses of `ioctl` accepted upstream. The kernel maintainers dislike `ioctl` because it makes kernel code and application code too interdependent, and it is hard to keep both of them in step across kernel versions and architectures.
- `sysfs`: This is the preferred way now, a good example being the GPIO interface described earlier. The advantages are that it is self-documenting, so long as you choose descriptive names for the files. It is also scriptable because the file contents are ASCII strings. On the other hand, the requirement for each file to contain a single value makes it hard to achieve atomicity if you need to change more than one value at a time. For example, if you want to set two values and then initiate an action, you would need to write to three files: two for the inputs and a third to trigger the action. Even then, there is no guarantee that the other two files have not been changed by someone else. Conversely, `ioctl` passes all its arguments in a structure in a single function call.

- `mmap`: You can get direct access to kernel buffers and hardware registers by mapping kernel memory into user-space, bypassing the kernel. You may still need some kernel code to handle interrupts and DMA. There is a subsystem that encapsulates this idea, known as `uio`, short for user I/O. There is more documentation in `Documentation/DocBook/uio-howto`, and there are example drivers in `drivers/uio`.
- `sigio`: You can send a signal from a driver using the kernel function `kill_fasync()` to notify applications of an event such as input becoming ready or an interrupt being received. By convention, signal `SIGIO` is used, but it could be anyone. You can see some examples in the UIO driver, `drivers/uio/uio.c`, and in the RTC driver, `drivers/char/rtc.c`. The main problem is that it is difficult to write reliable signal handlers and so it remains a little-used facility.
- `debugfs`: This is another pseudo filesystem that represents kernel data as files and directories, similar to `proc` and `sysfs`. The main distinction is that `debugfs` must not contain information that is needed for the normal operation of the system; it is for debug and trace information only. It is mounted as `mount -t debugfs debug /sys/kernel/debug`.

There is a good description of `debugfs` in the kernel documentation, `Documentation/filesystems/debugfs.txt`.

- `proc`: The `proc` filesystem is deprecated for all new code unless it relates to processes, which was the original intended purpose for the filesystem. However, you can use `proc` to publish any information you choose. And, unlike `sysfs` and `debugfs`, it is available to non-GPL modules.
- `netlink`: This is a socket protocol family. `AF_NETLINK` creates a socket that links kernel space to user-space. It was originally created so that network tools could communicate with the Linux network code to access the routing tables and other details. It is also used by `udev` to pass events from the kernel to the `udev` daemon. It is very rarely used in general device drivers.

There are many examples of all of the preceding filesystem in the kernel source code and you can design really interesting interfaces to your driver code. The only universal rule is the *principle of least astonishment*. In other words, application writers using your driver should find that everything works in a logical way with no quirks or oddities.

Anatomy of a device driver

It's time to draw some threads together by looking at the code for a simple device driver.

The source code is provided for a device driver named `dummy` which creates four devices that are accessed through `/dev/dummy0` to `/dev/dummy3`. This is the complete code for the driver:

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/device.h>
#define DEVICE_NAME "dummy"
#define MAJOR_NUM 42
#define NUM_DEVICES 4

static struct class *dummy_class;
static int dummy_open(struct inode *inode, struct file *file)
{
    pr_info("%s\n", __func__);
    return 0;
}

static int dummy_release(struct inode *inode, struct file *file)
{
    pr_info("%s\n", __func__);
    return 0;
}

static ssize_t dummy_read(struct file *file,
    char *buffer, size_t length, loff_t * offset)
{
    pr_info("%s %u\n", __func__, length);
    return 0;
}

static ssize_t dummy_write(struct file *file,
    const char *buffer, size_t length, loff_t * offset)
{
    pr_info("%s %u\n", __func__, length);
    return length;
}
```

```
struct file_operations dummy_fops = {
    .owner = THIS_MODULE,
    .open = dummy_open,
    .release = dummy_release,
    .read = dummy_read,
    .write = dummy_write,
};

int __init dummy_init(void)
{
    int ret;
    int i;
    printk("Dummy loaded\n");
    ret = register_chrdev(MAJOR_NUM, DEVICE_NAME, &dummy_fops);
    if (ret != 0)
        return ret;
    dummy_class = class_create(THIS_MODULE, DEVICE_NAME);
    for (i = 0; i < NUM_DEVICES; i++) {
        device_create(dummy_class, NULL,
                      MKDEV(MAJOR_NUM, i), NULL, "dummy%d", i);
    }
    return 0;
}

void __exit dummy_exit(void)
{
    int i;
    for (i = 0; i < NUM_DEVICES; i++) {
        device_destroy(dummy_class, MKDEV(MAJOR_NUM, i));
    }
    class_destroy(dummy_class);
    unregister_chrdev(MAJOR_NUM, DEVICE_NAME);
    printk("Dummy unloaded\n");
}

module_init(dummy_init);
module_exit(dummy_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Chris Simmonds");
MODULE_DESCRIPTION("A dummy driver");
```

At the end of the code, the macros `module_init` and `module_exit` specify the functions to be called when the module is loaded and unloaded. The other three add some basic information about the module which can be retrieved from the compiled kernel module using the `modinfo` command.

When the module is loaded, the `dummy_init()` function is called.

You can see the point at which it becomes a character device by calling `register_chrdev`, passing a pointer to `struct file_operations` containing pointers to the four functions that the driver implements. While `register_chrdev` tells the kernel that there is a driver with a major number of 42, it doesn't say anything about the type of driver, and so it will not create an entry in `/sys/class`. Without an entry in `/sys/class`, the device manager cannot create device nodes. So, the next few lines of code create a device class, `dummy`, and four devices of that class called `dummy0` to `dummy3`. The result is the `/sys/class/dummy` directory, containing the `dummy0` to `dummy3` subdirectories, each containing a file, `dev`, with the major and minor numbers of the device. This is all that a device manager needs to create device nodes, `/dev/dummy0` to `/dev/dummy3`.

The `exit` function has to release the resources claimed by the `init` function which, here, means freeing up the device class and major number.

The file operation for this driver are implemented by `dummy_open()`, `dummy_read()`, `dummy_write()`, and `dummy_release()`, and are called when a user space program calls `open(2)`, `read(2)`, `write(2)`, and `close(2)`. They just print a kernel message so that you can see that they were called. You can demonstrate this from the command line using the `echo` command:

```
# echo hello > /dev/dummy0

[ 6479.741192] dummy_open
[ 6479.742505] dummy_write 6
[ 6479.743008] dummy_release
```

In this case, the messages appear because I was logged on to the console, and kernel messages are printed to the console by default.

The full source code for this driver is less than 100 lines, but it is enough to illustrate how the linkage between a device node and driver code works, how the device class is created, allowing a device manager to create device nodes automatically when the driver is loaded, and how data is moved between user and kernel spaces. Next, you need to build it.

Compile and load

At this point you have some driver code that you want to compile and test on your target system. You can copy it into the kernel source tree and modify makefiles to build it, or you can compile it as a module out of tree. Let's start by building out of tree.

You need a simple makefile which uses the kernel build system to do the hard work:

```
LINUXDIR := $(HOME)/MELP/build/linux

obj-m := dummy.o
all:
    make ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf- \
        -C $(LINUXDIR) M=$(shell pwd)
clean:
    make -C $(LINUXDIR) M=$(shell pwd) clean
```

Set `LINUXDIR` to the directory of the kernel for your target device that you will be running the module on. The code `obj-m := dummy.o` will invoke the kernel build rule to take the source file, `dummy.c` and create kernel module, `dummy.ko`. Note that kernel modules are not binary compatible between kernel releases and configurations, the module will only load on the kernel it was compiled with.

The end result of the build is the kernel `dummy.ko` which you can copy to the target and load as shown in the next section.

If you want to build a driver in the kernel source tree, the procedure is quite simple. Choose a directory appropriate to the type of driver you have. The driver is a basic character device, so I would put `dummy.c` in `drivers/char`. Then, edit the makefile in that directory and add a line to build the driver unconditionally as a module, as follows:

```
obj-m += dummy.o
```

Or add the following line this to build it unconditionally as a built-in:

```
obj-y += dummy.o
```

If you want to make the driver optional, you can add a menu option to the `Kconfig` file and make the compilation conditional on the configuration option, as I described in *Chapter 4, Porting and Configuring the Kernel*, when describing the kernel configuration.

Loading kernel modules

You can load, unload and list modules using the simple `insmod`, `lsmod`, and `rmmmod` commands. Here they are shown loading the dummy driver:

```
# insmod /lib/modules/4.1.10/kernel/drivers/dummy.ko
# lsmod
dummy 1248 0 - Live 0xbff009000 (0)
# rmmmod dummy
```

If the module is placed in a subdirectory in `/lib/modules/<kernel release>`, as in the example, you can create a modules dependency database using the command `depmod`:

```
# depmod -a
# ls /lib/modules/4.1.10/
kernel           modules.builtin.bin  modules.order
modules.alias    modules.dep          modules.softdep
modules.alias.bin modules.dep.bin    modules.symbols
modules.builtin  modules.devname    modules.symbols.bin
```

The information in the `module.*` files is used by the command `modprobe` to locate a module by name rather than the full path. `modprobe` has many other features which are described in the manual.

The module dependency information is also used by device managers, `udev` in particular. When new hardware is detected, for example a new USB device, the `udevd` daemon is alerted and passed the vendor, and product IDs are read from the hardware. `udevd` scans the module dependency files looking for a module that has registered those IDs. If one is found, it is loaded using `modprobe`.

Discovering hardware configuration

The dummy driver demonstrates the structure of a device driver, but it lacks interaction with real hardware since it only manipulates memory structures. Device drivers are usually written to interact with hardware and part of that is being able to discover the hardware in the first place, bearing in mind that it may be at different addresses in different configurations.

In some cases, the hardware provides the information itself. Devices on a discoverable bus such as PCI or USB have a query mode which returns resource requirements and a unique identifier. The kernel matches the identifier and possibly other characteristics with the device drivers, and marries them up.

However, most of the hardware blocks on an SoC do not have such identifiers. You have to provide the information yourself in the form of a device tree or as C structures known as platform data.

In the standard driver model for Linux, device drivers register themselves with the appropriate subsystem: PCI, USB, open firmware (device tree), platform device, and so on. The registration includes an identifier and a callback function called a probe function that is called if there is a match between the ID of the hardware and the ID of the driver. For PCI and USB, the ID is based on the vendor and the product IDs of the devices, for device tree and platform devices, it is a name (an ASCII string).

Device trees

I gave you an introduction to device trees in *Chapter 3, All About Bootloaders*. Here, I want to show you how the Linux device drivers hook up with that information.

As an example, I will use the ARM Versatile board, `arch/arm/boot/dts/versatile-ab.dts`, for which the Ethernet adapter is defined here:

```
net@10010000 {  
    compatible = "smsc,lan91c111";  
    reg = <0x10010000 0x10000>;  
    interrupts = <25>;  
};
```

Platform data

In the absence of device tree support, there is a fallback method of describing hardware using C structures, known as platform data.

Each piece of hardware is described by `struct platform_device`, which has a name and a pointer to an array of resources. The type of the resource is determined by flags, which include the following:

- `IORESOURCE_MEM`: The physical address of a region of memory
- `IORESOURCE_IO`: The physical address or port number of IO registers
- `IORESOURCE_IRQ`: The interrupt number

Here is an example of the platform data for an Ethernet controller taken from `arch/arm/mach-versatile/core.c`, which has been edited for clarity:

```
#define VERSATILE_ETH_BASE      0x10010000
#define IRQ_ETH                  25
static struct resource smc91x_resources[] = {
    [0] = {
        .start      = VERSATILE_ETH_BASE,
        .end        = VERSATILE_ETH_BASE + SZ_64K - 1,
        .flags      = IORESOURCE_MEM,
    },
    [1] = {
        .start      = IRQ_ETH,
        .end        = IRQ_ETH,
        .flags      = IORESOURCE_IRQ,
    },
};

static struct platform_device smc91x_device = {
    .name       = "smc91x",
    .id         = 0,
    .num_resources = ARRAY_SIZE(smc91x_resources),
    .resource   = smc91x_resources,
};
```

It has a memory area of 64 KiB and an interrupt. The platform data has to be registered with the kernel, usually when the board is initialized:

```
void __init versatile_init(void)
{
    platform_device_register(&versatile_flash_device);
    platform_device_register(&versatile_i2c_device);
    platform_device_register(&smc91x_device);
    [...]
```

Linking hardware with device drivers

You have seen in the preceding section how an Ethernet adapter is described using a device tree and using platform data. The corresponding driver code is in `drivers/net/ethernet/smsc/smc91x.c` and it works with both the device tree and platform data. Here is the initialization code, once again edited for clarity:

```
static const struct of_device_id smc91x_match[] = {
    { .compatible = "smsc,lan91c94", },
    { .compatible = "smsc,lan91c111", },
```

```
{},
};

MODULE_DEVICE_TABLE(of, smc91x_match);
static struct platform_driver smc_driver = {
    .probe          = smc_drv_probe,
    .remove         = smc_drv_remove,
    .driver         = {
        .name   = "smc91x",
        .of_match_table = of_match_ptr(smc91x_match),
    },
};
static int __init smc_driver_init(void)
{
    return platform_driver_register(&smc_driver);
}
static void __exit smc_driver_exit(void) \
{
    platform_driver_unregister(&smc_driver);
}
module_init(smc_driver_init);
module_exit(smc_driver_exit);
```

When the driver is initialized, it calls `platform_driver_register()`, pointing to `struct platform_driver`, in which there is a callback to a probe function, a driver name, `smc91x`, and a pointer to `struct of_device_id`.

If this driver has been configured by the device tree, the kernel will look for a match between the `compatible` property in the device tree node and the string pointed to by the `compatible` structure element. For each match, it calls the `probe` function.

On the other hand, if it was configured through platform data, the `probe` function will be called for each match on the string pointed to by `driver.name`.

The `probe` function extracts information about the interface:

```
static int smc_drv_probe(struct platform_device *pdev)
{
    struct smc91x_platdata *pd = dev_get_platdata(&pdev->dev);
    const struct of_device_id *match = NULL;
    struct resource *res, *ires;
    int irq;

    res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    ires = platform_get_resource(pdev, IORESOURCE_IRQ, 0);
    [...]
```

```
    addr = ioremap(res->start, SMC_IO_EXTENT) ;
    irq = ires->start;
    [...]
}
```

The calls to `platform_get_resource()` extract the memory and `irq` information from either the device tree or the platform data. It is up to the driver to map the memory and install the interrupt handler. The third parameter, which is zero in both of the previous cases, comes into play if there is more than one resource of that particular type.

Device trees allow you to configure more than just basic memory ranges and interrupts, however. There is a section of code in the `probe` function that extracts optional parameters from the device tree. In this snippet, it gets the `register-io-width` property:

```
match = of_match_device(of_match_ptr(smc91x_match), &pdev->dev);
if (match) {
    struct device_node *np = pdev->dev.of_node;
    u32 val;
    [...]
    of_property_read_u32(np, "reg-io-width", &val);
    [...]
}
```

For most drivers, specific bindings are documented in [Documentation/devicetree/bindings](#). For this particular driver, the information is in [Documentation/devicetree/bindings/net/smsc911x.txt](#).

The main thing to remember here is that drivers should register a `probe` function and enough information for the kernel to call the `probe` as it finds matches with the hardware it knows about. The linkage between the hardware described by the device tree and the device driver is through the `compatible` property. The linkage between platform data and a driver is through the name.

Additional reading

The following resources have further information about the topics introduced in this chapter:

- *Linux Device Drivers, 4th edition*, by Jessica McKellar, Alessandro Rubini, Jonathan Corbet, and Greg Kroah-Hartman. This is not published at the time of writing, but if it is as good as the predecessor, it will be a good choice. However, the 3rd edition is too out of date to recommend.
- *Linux Kernel Development, 3rd edition* by Robert Love, Addison-Wesley Professional; (July 2, 2010) ISBN-10: 0672329468
- *Linux Weekly News*, lwn.net.

Summary

Device drivers have the job of handling devices, usually physical hardware but sometimes virtual interfaces, and presenting it to higher levels in a consistent and useful way. Linux device drivers fall into three broad categories: the character, the block, and the network. Of the three, the character driver interface is the most flexible and therefore, the most common. Linux drivers fit into a framework known as the driver model, which is exposed through sysfs. Pretty much the entire state of the devices and drivers is visible in /sys.

Each embedded system has its own unique set of hardware interfaces and requirements. Linux provides drivers for most standard interfaces, and by selecting the right kernel configuration, you can get the device operational very quickly. That leaves you with the non-standard components for which you will have to add your own device support.

In some cases, you can sidestep the issue by using generic drivers for GPIO, I2C, and so on, and write user space code to do the work. I recommend this as a starting point as it gives you the chance to get familiar with the hardware without writing kernel code. Writing kernel drivers is not particularly difficult but, if you do, you need to code carefully so as to not compromise the stability of the system.

I have talked about writing kernel driver code: if you go down that route, you will inevitably want to know how to check whether or not it is working correctly and detect any bugs. I will cover that topic in *Chapter 12, Debugging with GDB*.

The next chapter is all about user space initialization and the different options you have for the `init` program, from the simple BusyBox to the complex systemd.

9

Starting up - the init Program

I looked at how the kernel boots up to the point that it launches the first program, `init`, in *Chapter 4, Porting and Configuring the Kernel* and in *Chapter 5, Building a Root Filesystem* and *Chapter 6, Selecting a Build System*, I looked at creating root filesystems of varying complexity, all of which contained an `init` program. Now it is time to look at the `init` program in more detail and discover why it is so important to the rest of the system.

There are many possible implementations of `init`. I will describe the three main ones in this chapter: BusyBox `init`, System V `init`, and `systemd`. For each one, I will give an overview of how it works and the types of system it suits best. Part of that is balancing the trade-off between complexity and flexibility.

After the kernel has booted

We saw in *Chapter 4, Porting and Configuring the Kernel*, how the kernel bootstrap code seeks to find a root filesystem, either `initramfs` or a filesystem specified by `root=` on the kernel command line, and then to execute a program which, by default, is `/init` for `initramfs`, and `/sbin/init` for a regular filesystem. The `init` program has root privilege and since it is the first process to run, it has a process ID (PID) of 1. If, for some reason, `init` cannot be started, the kernel will panic.

The `init` program is the ancestor of all other processes, as shown here by the `pstree` command, which is part of the `psmisc` package in most distributions:

```
# pstree -gn
```

```
init(1) --> syslogd(63)
           | -klogd(66)
           | -dropbear(99)
           `-- sh(100) --- pstree(109)
```

The job of the `init` program is to take control of the system and set it running. It may be as simple as a shell command running a shell script – there is an example at the start of *Chapter 5, Building a Root Filesystem* – but, in the majority of cases, you will be using a dedicated `init` daemon. The tasks it has to perform are as follows:

- At boot, it starts daemon programs, configures system parameters and the other things needed to get the system into a working state.
- Optionally, it launches daemons, such as `getty` on terminals which allow a login shell.
- It adopts processes that become orphaned as a result of their immediate parent terminating and there being no other processes in the thread group.
- It responds to any of `init`'s immediate children terminating by catching the signal `SIGCHLD` and collecting the return value to prevent them becoming zombie processes. I will talk more about zombies in *Chapter 10, Learning About Processes and Threads*.
- Optionally, it restarts those daemons that have terminated.
- It handles system shutdown.

In other words, `init` manages the lifecycle of the system, from boot up to shutdown. The current thinking is that `init` is well placed to handle other runtime events such as new hardware and the loading and unloading of modules. This is what `systemd` does.

Introducing the init programs

The three `init` programs that you are most likely to encounter in embedded devices are BusyBox `init`, System V `init`, and `systemd`. Buildroot has options to build all three with BusyBox `init` as the default. The Yocto Project allows you to choose between System V `init` and `systemd`, with System V `init` the default.

The following table gives some metrics to compare the three:

	BusyBox init	System V init	systemd
Complexity	Low	Medium	High
Boot-up speed	Fast	Slow	Medium
Required shell	ash	ash or bash	None
Number of executables	0	4	50(*)
libc	Any	Any	glibc
Size (MiB)	0	0.1	34(*)

(*) Based on the Buildroot configuration of `system`.

Broadly speaking, there is an increase in flexibility and complexity as you go from BusyBox init to systemd.

BusyBox init

BusyBox has a minimal init program that uses a configuration file, `/etc/inittab`, to define rules to start programs at boot up and to stop them at shutdown. Usually, the actual work is done by shell scripts which, by convention, are placed in the `/etc/init.d` directory.

init begins by reading the configuration file, `/etc/inittab`. This contains a list of programs to run, one per line, with this format:

```
<id>::<action>:<program>
```

The role of these parameters is as follows:

- `id`: The controlling terminal for the command
- `action`: The conditions to run this command, as shown in the following paragraph
- `program`: The program to run

The actions are as follows:

- `sysinit`: Run the program when init starts, before any of the other types of actions.
- `respawn`: Run the program and restart it if it terminates. It is used to run a program as a daemon.

- `askfirst`: This is the same as `respawn`, but prints the message **Please press Enter to activate this console** to the console and runs the program after *Enter* has been pressed. It is used to start an interactive shell on a terminal without prompting for a user name or password.
- `once`: Run the program once but do not attempt to restart it if it terminates.
- `wait`: Run the program and wait for it to complete.
- `restart`: Run the program when `init` receives the signal `SIGHUP`, indicating that it should reload the `inittab` file.
- `ctrlaltdel`: Run the program when `init` receives the signal `SIGINT`, usually as a result of pressing *Ctrl + Alt + Del* on the console.
- `shutdown`: Run the program when `init` shuts down.

Here is a small example that mounts `proc` and `sysfs` and runs a shell on a serial interface:

```
null::sysinit:/bin/mount -t proc proc /proc
null::sysinit:/bin/mount -t sysfs sysfs /sys
console::askfirst:-/bin/sh
```

For simple projects in which you want to launch a small number of daemons and perhaps start a login shell on a serial terminal, it is easy to write the scripts manually, and this is appropriate if you are creating a **RYO (roll your own)** embedded Linux. However, you will find that hand-written `init` scripts rapidly become unmaintainable as the number of things to be configured increases. They tend not to be very modular and so need updating each time a new component is added.

Buildroot init scripts

Buildroot has been making effective use of BusyBox `init` for many years. Buildroot has two scripts in `/etc/init.d` named `rcS` and `rcK`. The first one starts at boot-up and iterates over all the scripts beginning with a capital `s` followed by two digits, and runs them in numerical order. These are the start scripts. The `rcK` script is run at shutdown and iterates over all the scripts beginning with a capital `k` followed by two digits, and runs them in numerical order. These are the kill scripts.

With this in place, it becomes easy for Buildroot packages to supply their own start and kill scripts, using the two digit number to impose the order in which they should be run, and so the system becomes extensible. If you are using Buildroot, this is transparent. If not, you could use it as a model for writing your own BusyBox `init` scripts.

System V init

This `init` program was inspired by the one from UNIX System V, and so dates back to the mid 1980s. The version most often found in Linux distributions was written initially by Miquel van Smoorenburg. Until recently, it was considered the way to boot Linux, obviously including embedded systems, and BusyBox `init` is a trimmed down version of System V `init`.

Compared to BusyBox `init`, System V `init` has two advantages. Firstly, the boot scripts are written in a well-known, modular format, making it easy to add new packages at build time or runtime. Secondly, it has the concept of runlevels, which allow a collection of programs to be started or stopped in one go, by switching from one runlevel to another.

There are 8 runlevels numbered from 0 to 6, plus S:

- **S:** Single user mode
- **0:** Halt the system
- **1 to 5:** General use
- **6:** Reboot the system

Levels 1 to 5 can be used as you please. On desktop Linux distributions, they are conventionally assigned as follows:

- **1:** Single user
- **2:** Multi-user with no network configuration
- **3:** Multi-user with network configuration
- **4:** Not used
- **5:** Multi-user with graphical login

The `init` program starts the default runlevel given by the `initdefault` line in `/etc/inittab`. You can change the runlevel at runtime using the command `telinit [runlevel]` which sends a message to `init`. You can find the current runlevel, and the previous one, by using the `runlevel` command. Here is an example:

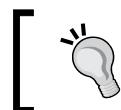
```
# runlevel
N 5
# telinit 3
INIT: Switching to runlevel: 3
# runlevel
5 3
```

On the first line, the output from `runlevel` is `N 5`, meaning that there is no previous runlevel because the `runlevel` has not changed since booting, and the current `runlevel` is 5. After changing the `runlevel`, the output is `5 3` showing that there has been a transition from 5 to 3. The `halt` and `reboot` commands switch to runlevels of 0 and 6 respectively. You can override the default `runlevel` by giving a different one on the kernel command line as a single digit from 0 to 6, or `s` for single user mode. For example, to force the `runlevel` to be for a single user, you would append `s` to the kernel command line and it would look something like this:

```
console=ttyAMA0 root=/dev/mmcblk1p2 s
```

Each runlevel has a number of scripts that stop things, called `kill` scripts, and another group that starts things, the `start` scripts. When entering a new runlevel, `init` first runs the `kill` scripts and then the `start` scripts. Running daemons which have neither a `start` script nor a `kill` script in the new runlevel are sent a `SIGTERM` signal. In other words, the default action on switching runlevel is to terminate the daemons unless told to do otherwise.

In truth, runlevels are not used that much in embedded Linux: most devices simply boot to the default `runlevel` and stay there. I have a feeling that it is partly because most people are not aware of them.



Runlevels are a simple and convenient way to switch between modes, for example, from production to maintenance mode.



System V `init` is an option in Buildroot and the Yocto Project. In both cases, the `init` scripts have been stripped of any bash specifics, so they work with the BusyBox `ash` shell. However, Buildroot cheats by replacing the BusyBox `init` program with SystemV `init` and adding `inittab` that mimics the behavior of BusyBox. Buildroot does not implement runlevels except that switching to levels 0 or 6 halts or reboots the system.

Next, let's look at some of the details. The following examples are taken from the fido version of the Yocto Project. Other distributions may implement the `init` scripts a little differently.

inittab

The `init` program begins by reading `/etc/inittab`, which contains entries that define what happens at each `runlevel`. The format is an extended version of the BusyBox `inittab` that I described in the preceding section, which is not surprising because BusyBox borrowed it from System V in the first place!

The format of each line in `inittab` is as follows:

```
id:runlevels:action:process
```

The fields are shown here:

- `id`: A unique identifier of up to four characters.
- `runlevels`: The runlevels for which this entry should be executed.
(This was left blank in the BusyBox `inittab`)
- `action`: One of the keywords given as follows.
- `process`: The command to run.

The actions are the same as for BusyBox `init`: `sysinit`, `respawn`, `once`, `wait`, `restart`, `ctrlaltdel`, and `shutdown`. However, System V `init` does not have `askfirst`, which is specific to BusyBox.

As an example, this is the complete `inttab` supplied by the Yocto Project target `core-image-minimal`:

```
# /etc/inittab: init(8) configuration.
# $Id: inittab,v 1.91 2002/01/25 13:35:21 miquels Exp $

# The default runlevel.
id:5:initdefault:

# Boot-time system configuration/initialization script.
# This is run first except when booting in emergency (-b) mode.
si::sysinit:/etc/init.d/rcS

# What to do in single-user mode.
~~:S:wait:/sbin/sulogin
# /etc/init.d executes the S and K scripts upon change
# of runlevel.
#
# Runlevel 0 is halt.
# Runlevel 1 is single-user.
# Runlevels 2-5 are multi-user.
# Runlevel 6 is reboot.

10:0:wait:/etc/init.d/rc 0
11:1:wait:/etc/init.d/rc 1
12:2:wait:/etc/init.d/rc 2
13:3:wait:/etc/init.d/rc 3
14:4:wait:/etc/init.d/rc 4
```

Starting Up - the init Program

```
15:5:wait:/etc/init.d/rc 5
16:6:wait:/etc/init.d/rc 6
# Normally not reached, but fallthrough in case of emergency.
z6:6:respawn:/sbin/sulogin
AMA0:12345:respawn:/sbin/getty 115200 ttyAMA0
# /sbin/getty invocations for the runlevels.
#
# The "id" field MUST be the same as the last
# characters of the device (after "tty").
#
# Format:
# <id>:<runlevels>:<action>:<process>
#
1:2345:respawn:/sbin/getty 38400 tty1
```

The first entry, `id:5:initdefault`, sets the default runlevel to 5. The next entry, `si::sysinit:/etc/init.d/rcS`, runs the script `rcS` at boot up. There will be more about this later. A little further on, there is a group of six entries beginning with `10:0:wait:/etc/init.d/rc 0`. They run the script `/etc/init.d/rc` each time there is a change in the runlevel: this script is responsible for processing the start and kill scripts. There is an entry for runlevel `s` which runs the single-user login program.

Towards the end of `inittab`, there are two entries that run a `getty` daemon to generate a login prompt on the devices `/dev/ttyAMA0` and `/dev/ttym1` when entering runlevels 1 through to 5, thereby allowing you to log on and get an interactive shell:

```
AMA0:12345:respawn:/sbin/getty 115200 ttyAMA0
1:2345:respawn:/sbin/getty 38400 ttym1
```

The device `ttyAMA0` is the serial console on the ARM Versatile board we are emulating with QEMU, it will be different for other development boards. `Tty1` is a virtual console which is often mapped to a graphical screen if you have built your kernel with `CONFIG_FRAMEBUFFER_CONSOLE` or `VGA_CONSOLE`. Desktop Linux usually spawns six `getty` processes on virtual terminals 1 to 6, which you can select with the key combination `Ctrl + Alt + F1` through `Ctrl + Alt + F6`, with virtual terminal 7 reserved for the graphical screen. Virtual terminals are seldom used on embedded devices.

The script `/etc/init.d/rcS` that is run by the `sysinit` entry does little more than enter runlevel `s`:

```
#!/bin/sh

[...]
exec /etc/init.d/rc S
```

Hence, the first run level entered is S, followed by the default runlevel of 5. Note that runlevel S is not recorded and is never displayed as a prior runlevel by the runlevel command.

The init.d scripts

Each component that needs to respond to a runlevel change has a script in /etc/init.d to perform that change. The script should expect two parameters: start and stop. I will give an example of this later.

The runlevel handling script, /etc/init.d/rc, takes the runlevel it is switching to as a parameter. For each runlevel, there is a directory named rc<runlevel>.d:

```
# ls -d /etc/rc*
/etc/rc0.d /etc/rc2.d /etc/rc4.d /etc/rc6.d
/etc/rc1.d /etc/rc3.d /etc/rc5.d /etc/rcS.d
```

There you will find a set of scripts beginning with a capital S followed by two digits and you may also find scripts beginning with a capital K. These are start and kill scripts: Buildroot uses the same idea, borrowed from here:

```
# ls /etc/rc5.d
S01networking S20hwclock.sh S99rmnologin.sh S99stop-bootlogd
S15mountnfs.sh S20syslog
```

These are in fact symbolic links back to the appropriate script in init.d. The rc script runs all the scripts beginning with a K first, adding the stop parameter , and then runs those beginning with an S adding the start parameter . Once again, the two digit code is there to impart the order in which the scripts should run.

Adding a new daemon

Imagine that you have a program named simpleserver which is written as a traditional Unix daemon, in other words, it forks and runs in the background. You will need an init.d script like this:

```
#!/bin/sh

case "$1" in
start)
echo "Starting simpleserver"
```

Starting Up - the init Program

```
start-stop-daemon -S -n simpleserver -a /usr/bin/simpleserver
;;
stop)
    echo "Stopping simpleserver"
    start-stop-daemon -K -n simpleserver
;;
*)
    echo "Usage: $0 {start|stop}"
    exit 1
esac

exit 0
```

Start-stop-daemon is a helper function that makes it easier to manipulate background processes such as this. It originally came from the Debian installer package, dpkg, but most embedded systems use the one from BusyBox. It starts the daemon with the `-S` parameter, making sure that there is never more than one instance running at any one time and it finds the daemon by name with `-K` and sends a signal, SIGTERM, by default. Place this script in `/etc/init.d/simpleserver` and make it executable.

Then, add symlinks from each of the run levels that you want to run this program from, in this case, only the default runlevel, 5:

```
# cd /etc/init.d/rc5.d
# ln -s ../init.d/simpleserver S99simpleserver
```

The number 99 means that this will be one of the last programs to be started. Bear in mind that there may be other links beginning S99, in which case the `rc` script will just run them in lexical order.

It is rare in embedded devices to have to worry too much about shutdown operations, but if there is something that needs to be done, add `kill` symlinks to levels 0 and 6:

```
# cd /etc/init.d/rc0.d
# ln -s ../init.d/simpleserver K01simpleserver
# cd /etc/init.d/rc6.d
# ln -s ../init.d/simpleserver K01simpleserver
```

Starting and stopping services

You can interact with the scripts in `/etc/init.d` by calling them directly with, for example, the `syslog` script which controls the `syslogd` and `klogd` daemons:

```
# /etc/init.d/syslog --help
Usage: syslog { start | stop | restart }

# /etc/init.d/syslog stop
Stopping syslogd/klogd: stopped syslogd (pid 198)
stopped klogd (pid 201)
done

# /etc/init.d/syslog start
Starting syslogd/klogd: done
```

All scripts implement `start` and `stop` and should implement `help`. Some implement `status` as well, which will tell you whether the service is running or not. Mainstream distributions that still use System V init have a command named `service` to start and stop services and hide the details of calling the scripts directly.

systemd

`systemd` defines itself as a system and service manager. The project was initiated in 2010 by Lennart Poettering and Kay Sievers to create an integrated set of tools for managing a Linux system including an `init` daemon. It also includes device management (`udev`) and logging, among other things. Some would say that it is not just an `init` program, it is a way of life. It is state of the art, and still evolving rapidly. `systemd` is common on desktop and server Linux distributions, and is becoming popular on embedded Linux systems too, especially on more complex devices. So, how is it better than System V `init` for embedded systems?

- Configuration is simpler and more logical (once you understand it), rather than the sometimes convoluted shell scripts of System V `init`, `systemd` has unit configuration files to set parameters
- There are explicit dependencies between services rather than a two digit code that merely sets the sequence in which the scripts are run
- It is easy to set the permissions and resource limits for each service, which is important for security
- `systemd` can monitor services and restart them if needed
- There are watchdogs for each service and for `systemd` itself
- Services are started in parallel, reducing boot time

A complete description of `systemd` is neither possible nor appropriate here. As with System V `init`, I will focus on embedded use-cases, with examples based on the configuration produced by Yocto Fido, which has `systemd` version 219. I will give a quick overview and then show you some specific examples.

Building `systemd` with the Yocto Project and Buildroot

The default `init` in Yocto Fido is System V. To select `systemd`, add these lines to your configuration, for example, in `conf/local.conf`:

```
DISTRO_FEATURES_append = " systemd"  
VIRTUAL-RUNTIME_init_manager = "systemd"
```

Note that the leading space is important! Then rebuild.

Buildroot has `systemd` as the third `init` option. It requires glibc as the C library, and kernel version 3.7 or later with a particular set of configuration options enabled. There is a complete list of dependencies in the `README` file in the top level of the `systemd` source code.

Introducing targets, services, and units

Before I describe how `systemd init` works, I need to introduce these three key concepts.

Firstly, a target is a group of services, similar to, but more general than, a SystemV `runlevel`. There is a default target which is the group of services that are started at boot time.

Secondly, a service is a daemon that can be started and stopped, very much like a SystemV service.

Finally, a unit is a configuration file that describes a target, a service, and several other things. Units are text files that contain properties and values.

You can change states and find out what is going on by using the `systemctl` command.

Units

The basic item of configuration is the unit file. Unit files are found in three different places:

- `/etc/systemd/system`: Local configuration
- `/run/systemd/system`: Runtime configuration
- `/lib/systemd/system`: Distribution-wide configuration

When looking for a unit, `systemd` searches the directories in that order, stopping as soon as it finds a match, allowing you to override the behavior of a distribution-wide unit by placing a unit of the same name in `/etc/systemd/system`. You can disable a unit completely by creating a local file that is empty or linked to `/dev/null`.

All unit files begin with a section marked `[Unit]` which contains basic information and dependencies, for example:

```
[Unit]
Description=D-Bus System Message Bus
Documentation=man:dbus-daemon(1)
Requires=dbus.socket
```

Unit dependencies are expressed through `Requires`, `Wants`, and `Conflicts`:

- `Requires`: A list of units that this unit depends on, which is started when this unit is started
- `Wants`: A weaker form of `Requires`: the units listed are started but the current unit is not stopped if any of them fail
- `Conflicts`: A negative dependency: the units listed are stopped when this one is started and, conversely, if one of them is started, this one is stopped

Processing the dependencies produces a list of units that should be started (or stopped). The keywords `Before` and `After` determine the order in which they are started. The order of stopping is just the reverse of the start order:

- `Before`: This unit should be started before the units listed
- `After`: This unit should be started after the units listed

In the following example, the `After` directive makes sure that the web server is started after the network:

```
[Unit]
Description=Lighttpd Web Server
After=network.target
```

In the absence of `Before` or `After` directives, the units will be started or stopped in parallel with no particular ordering.

Services

A service is a daemon that can be started and stopped, equivalent to a System V service. A service is a type of unit file with a name ending in `.service`, for example, `lighttpd.service`.

A service unit has a `[Service]` section that describes how it should be run. Here is the relevant section from `lighttpd.service`:

```
[Service]
ExecStart=/usr/sbin/lighttpd -f /etc/lighttpd/lighttpd.conf -D
ExecReload=/bin/kill -HUP $MAINPID
```

These are the commands to run when starting the service and restarting it. There are many more configuration points you can add in here, so refer to the man page for `systemd.service`.

Targets

A target is another type of unit which groups services (or other types of unit). It is a type of unit that only has dependencies. Targets have names ending in `.target`, for example, `multi-user.target`. A target is a desired state, which performs the same role as System V runlevels.

How systemd boots the system

Now we can see how `systemd` implements the bootstrap. `systemd` is run by the kernel as a result of `/sbin/init` being symbolically linked to `/lib/systemd/systemd`. It runs the default target, `default.target`, which is always a link to a desired target such as `multi-user.target` for a text login or `graphical.target` for a graphical environment. For example, if the default target is `multi-user.target`, you will find this symbolic link:

```
/etc/systemd/system/default.target -> /lib/systemd/system/multi-user.target
```

The default target may be overridden by passing `system.unit=<new target>` on the kernel command line. You can use `systemctl` to find out the default target, as shown here:

```
# systemctl get-default
multi-user.target
```

Starting a target such as `multi-user.target` creates a tree of dependencies that bring the system into a working state. In a typical system, `multi-user.target` depends on `basic.target`, which depends on `sysinit.target`, which depends on the services that need to be started early. You can print a graph using `systemctl list-dependencies`.

You can also list all the services and their current state using `systemctl list-units --type service`, and the same for targets using `systemctl list-units --type target`.

Adding your own service

Using the same `simpleserver` example as before, here is a service unit:

```
[Unit]
Description=Simple server

[Service]
Type=forking
ExecStart=/usr/bin/simpleserver

[Install]
WantedBy=multi-user.target
```

The `[Unit]` section only contains a description so that it shows up correctly when listed using `systemctl` and other commands. There are no dependencies; as I said, it is very simple.

The `[Service]` section points to the executable, and has a flag to indicate that it forks. If it were even simpler and ran in the foreground, `systemd` would do the daemonizing for us and `Type=forking` would not be needed.

The `[Install]` section makes it dependent on `multi-user.target` so that our server is started when the system goes into multi-user mode.

Once the unit is saved in `/etc/systemd/system/simpleserver.service`, you can start and stop it using the `systemctl start simpleserver` and `systemctl stop simpleserver` commands. You can use this command to find its current status:

```
# systemctl status simpleserver
simpleserver.service - Simple server
Loaded: loaded (/etc/systemd/system/simpleserver.service;
disabled)
Active: active (running) since Thu 1970-01-01 02:20:50 UTC; 8s
ago
```

Starting Up - the init Program

```
Main PID: 180 (simpleserver)
CGroup: /system.slice/simpleserver.service
└─180 /usr/bin/simpleserver -n
```

```
Jan 01 02:20:50 qemuarm systemd[1]: Started Simple server.
```

At this point, it will only start and stop on command, as shown. To make it persistent, you need to add a permanent dependency to a target. That is the purpose of the [Install] section in the unit, it says that when this service is enabled it will become dependent on multi-user.target, and so will be started at boot time. You enable it using `systemctl enable`, like this:

```
# systemctl enable simpleserver
Created symlink from /etc/systemd/system/multi-
user.target.wants/simpleserver.service to
/etc/systemd/system/simpleserver.service.
```

Now you can see how dependencies are added at runtime without having to edit any unit files. A target can have a directory named `<target_name>.target.wants` which can contain links to services. This is exactly the same as adding the dependent unit to the [Wants] list in the target. In this case, you will find that this link has been created:

```
/etc/systemd/system/multi-user.target.wants/simpleserver.service
/etc/systemd/system/simpleserver.service
```

If this were an important service you might want to restart if it failed. You can accomplish that by adding this flag to the [Service] section:

```
Restart=on-abort
```

Other options for `Restart` are `on-success`, `on-failure`, `on-abnormal`, `on-watchdog`, `on-abort`, or `always`.

Adding a watchdog

Watchdogs are a common requirement in embedded devices: you need to take action if a critical service stops working, usually by resetting the system. On most embedded SoCs, there is a hardware watchdog which can be accessed via the `/dev/watchdog` device node. The watchdog is initialized with a timeout at boot and then must be reset within that period, otherwise the watchdog will be triggered and the system will reboot. The interface with the watchdog driver is described in the kernel source in `Documentation/watchdog`, and the code for the drivers is in `drivers/watchdog`.

A problem arises if there are two or more critical services that need to be protected by a watchdog. `systemd` has a useful feature that distributes the watchdog between multiple services.

`systemd` can be configured to expect a regular keepalive call from a service and take action if it is not received, in other words, a per-service software watchdog. For this to work, you have to add code to the daemon to send the keepalive messages. It needs to check for a non-zero value in the `WATCHDOG_USEC` environment variable and then call `sd_notify(false, "WATCHDOG=1")` within that time (a period of half of the watchdog timeout is recommended). There are examples in the `systemd` source code.

To enable the watchdog in the service unit, add something like this to the [Service] section:

```
WatchdogSec=30s
Restart=on-watchdog
StartLimitInterval=5min
StartLimitBurst=4
StartLimitAction=reboot-force
```

In this example, the service expects a keepalive every 30 seconds. If it fails to be delivered, the service will be restarted, but if it is restarted more than four times in five minutes, `systemd` will force an immediate reboot. Once again, there is a full description of these settings in the `systemd` manual.

A watchdog like this takes care of individual services, but what if `systemd` itself fails, or the kernel crashes, or the hardware locks up. In those cases, we need to tell `systemd` to use the watchdog driver: just add `RuntimeWatchdogSec>NN` to `/etc/systemd/system.conf`. `systemd` will reset the watchdog within that period, and so the system will reset if `systemd` fails for some reason.

Implications for embedded Linux

`systemd` has a lot of features that are useful in embedded Linux, including many that I have not mentioned in this brief description such as resource control using slices (see the man page for `systemd.slice(5)` and `systemd.resource-control(5)`), device management (`udev(7)`) and system logging facilities (`journald(5)`).

You have to balance that with its size: even with a minimal build of just the core components, `systemd`, `udevd`, and `journald`, it is approaching 10 MiB of storage, including the shared libraries.

You also have to keep in mind that `systemd` development follows the kernel closely, so it will not work on a kernel more than a year or two older than the release of `systemd`.

Further reading

The following resource has further information about topics introduced in this chapter:

- *systemd system and Service Manager*: <http://www.freedesktop.org/wiki/Software/systemd/> (there are a lot of useful links at the bottom of that page)

Summary

Every Linux device needs an `init` program of some kind. If you are designing a system which only has to launch a small number of daemons at startup and remains fairly static after that, then BusyBox `init` is sufficient for your needs. It is usually a good choice if you are using Buildroot as the build system.

If, on the other hand, you have a system that has complex dependencies between services at boot time or runtime, and you have the storage space, then `systemd` would be the best choice. Even without the complexity, `systemd` has some useful features in the way it handles watchdogs, remote logging, and so on, so you should certainly give it a serious thought.

It is hard to make a case for System V `init` on its own merits, since it has few advantages over the simple BusyBox `init`. It will live on for a long time nevertheless, just because it is there. For example, if you are building using the Yocto Project and you decide against `systemd` then System V `init` is the alternative.

In terms of reducing boot time, `systemd` is faster than System V `init` for a similar workload. However, if you are looking for a very fast boot, nothing can beat a simple BusyBox `init` with minimal boot scripts.

This chapter is about one very important process, `init`. In the next chapter, I will describe what a process really is, how it relates to threads, how they cooperate, and how they are scheduled. Understanding these things is important if you want to create a robust and maintainable embedded system.

10

Learning About Processes and Threads

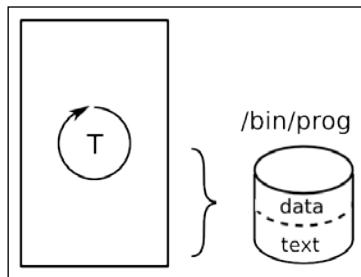
In the preceding chapters, we have considered the various aspects of creating an embedded Linux platform. Now it is time to start looking at how you can use the platform to create a working device. In this chapter, I will talk about the implications of the Linux process model and how it encompasses multi-threaded programs. I will look at the pros and cons of using single-threaded and multi-threaded processes. I will also look at scheduling and differentiate between timeshare and real-time scheduling policies.

While these topics are not specific to embedded computing, it is important for a designer of an embedded device to have an overview of these topics. There are many good reference works on the subject, some of which I reference at the end of the chapter, but in general, they do not consider the embedded use cases. In consequence, I will be concentrating on the concepts and design decisions rather than on the function calls and code.

Process or thread?

Many embedded developers who are familiar with **real-time operating systems (RTOS)** consider the Unix process model to be cumbersome. On the other hand, they see a similarity between an RTOS task and a Linux thread and they have a tendency to transfer an existing design using a one-to-one mapping of RTOS tasks to threads. I have, on several occasions, seen designs in which the entire application is implemented with one process containing 40 or more threads. I want to spend some time considering if this is a good idea or not. Let's begin with some definitions.

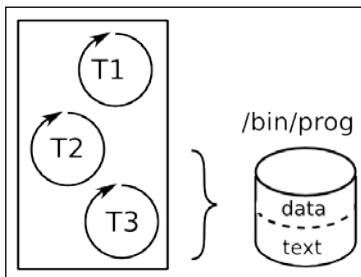
A process is a memory address space and a thread of execution, as shown in the following diagram. The address space is private to the process and so threads running in different processes cannot access it. This memory separation is created by the memory management subsystem in the kernel, which keeps a memory page mapping for each process and re-programs the memory management unit on each context switch. I will describe how this works in detail in *Chapter 11, Managing Memory*. Part of the address space is mapped to a file which contains the code and static data that the program is running:



As the program runs, it will allocate resources such as stack space, heap memory, references to files, and so on. When the process terminates, these resources are reclaimed by the system: all the memory is freed up and all the file descriptors are closed.

Processes can communicate with each other using **inter process communication (IPC)** such as local sockets. I will talk about IPC later on.

A thread is a thread of execution within a process. All processes begin with one thread that runs the `main()` function and is called the main thread. You can create additional threads using the POSIX threads function `pthread_create(3)`, causing additional threads to execute in the same address space, as shown in the following diagram. Being in the same process, they share resources with each other. They can read and write the same memory and use the same file descriptors, and so communication between threads is easy, so long as you take care of the synchronization and locking issues:



So, based on these brief details, you could imagine two extreme designs for a hypothetical system with 40 RTOS tasks being ported to Linux.

You could map tasks to processes, and have 40 individual programs communicating through IPC, for example with messages sent through sockets. You would greatly reduce memory corruption problems since the main thread running in each process is protected from the others, and you would reduce resource leakage since each process is cleaned up after it exits. However, the message interface between processes is quite complex and, where there is tight cooperation between a group of processes, the number of messages might be large and so become a limiting factor in the performance of the system. Furthermore, any one of the 40 processes may terminate, perhaps because of a bug causing it to crash, leaving the other 39 to carry on. Each process would have to handle the case that its neighbors are no longer running and recover gracefully.

At the other extreme, you could map tasks to threads and implement the system as a single process containing 40 threads. Cooperation becomes much easier because they share the same address space and file descriptors. The overhead of sending messages is reduced or eliminated and context switches between threads are faster than between processes. The downside is that you have introduced the possibility of one task corrupting the heap or the stack of another. If any one of the threads encounters a fatal bug, the whole process will terminate, taking all the threads with it. Finally, debugging a complex multi-threaded process can be a nightmare.

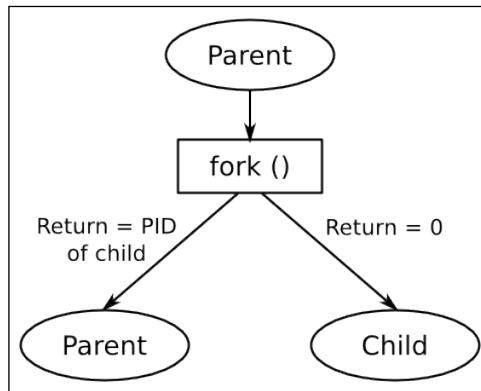
The conclusion you should draw is that neither design is ideal, and that there is a better way. But before we get to that point, I will delve a little more deeply into the APIs and the behavior of processes and threads.

Processes

A process holds the environment in which threads can run: it holds the memory mappings, the file descriptors, the user and group IDs, and more. The first process is the `init` process, which is created by the kernel during boot and has a PID of one. Thereafter, processes are created by duplication in an operation known as forking.

Creating a new process

The POSIX function to create a process is `fork(2)`. It is an odd function because, for each successful call, there are two returns: one in the process that made the call, known as the parent, and one in the newly created process, known as the child as shown in the following diagram:



Immediately after the call, the child is an exact copy of the parent, it has the same stack, the same heap, the same file descriptors, and executes the same line of code, the one following `fork(2)`. The only way the programmer can tell them apart is by looking at the return value of `fork`: it is zero for the child and greater than zero for the parent. Actually, the value returned in the parent is the PID of the newly created child process. There is a third possibility, which is that the return is negative, meaning that the `fork` call failed and there is still only one process.

Although the two processes are initially identical, they are in separate address spaces. Changes made to a variable by one will not be seen by the other. Under the hood, the kernel does not make a physical copy of the parent's memory, which would be quite a slow operation and consume memory unnecessarily. Instead, the memory is shared but marked with a **copy-on-write (CoW)** flag. If either parent or child modifies this memory, the kernel first makes a copy and then writes to the copy. This has the benefit of an efficient `fork` function while retaining the logical separation of process address spaces. I will discuss CoW in *Chapter 11, Managing Memory*.

Terminating a process

A process may be stopped voluntarily by calling the `exit(3)` function or, involuntarily, by receiving a signal that is not handled. One signal in particular, `SIGKILL`, cannot be handled and so will always kill a process. In all cases, terminating the process will stop all threads, close all file descriptors, and release all memory. The system sends a signal, `SIGCHLD`, to the parent so that it knows this has happened.

Processes have a return value which is composed of either the argument to `exit(3)`, if it terminated normally, or the signal number if it was killed. The chief use for this is in shell scripts: it allows you to test the return from a program. By convention, 0 indicates success and other values indicate a failure of some sort.

The parent can collect the return value with the `wait(2)` or `waitpid(2)` functions. This causes a problem: there will be a delay between a child terminating and its parent collecting the return value. In that period, the return value must be stored somewhere, and the PID number of the now dead process cannot be reused. A process in this state is a *zombie*, state Z in `ps` or `top`. So long as the parent calls `wait(2)` or `waitpid(2)`, whenever it is notified of a child's termination (by means of the `SIGCHLD` signal, see *Linux System Programming*, by Robert Love, O'Reilly Media or *The Linux Programming Interface*, by Michael Kerrisk, No Starch Press for details of handling signals), zombies exist for too short a time to show up in process listings. They will become a problem if the parent fails to collect the return value because you will not be able to create any more processes.

Here is a simple example, showing process creation and termination:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main(void)
{
    int pid;
    int status;
    pid = fork();
    if (pid == 0) {
        printf("I am the child, PID %d\n", getpid());
        sleep(10);
        exit(42);
    } else if (pid > 0) {
        printf("I am the parent, PID %d\n", getpid());
        wait(&status);
```

```
    printf("Child terminated, status %d\n",
           WEXITSTATUS(status));
} else
    perror("fork:");
return 0;
}
```

The `wait(2)` function blocks until a child process exits and stores the exit status. When you run it, you see something like this:

```
I am the parent, PID 13851
I am the child, PID 13852
Child terminated with status 42
```

The child process inherits most of the attributes of the parent, including the user and group IDs (UID and GID), all open file descriptors, signal handling, and scheduling characteristics.

Running a different program

The `fork` function creates a copy of a running program, but it does not run a different program. For that, you need one of the `exec` functions:

```
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg,
           ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[],
            char *const envp[]);
```

Each takes a path to the program file to load and run. If the function succeeds, the kernel discards all the resources of the current process, including memory and file descriptors, and allocates memory to the new program being loaded. When the thread that called `exec*` returns, it returns not to the line of code after the call, but to the `main()` function of the new program. Here is an example of a command launcher: it prompts for a command, for example, `/bin/ls`, and forks and executes the string you enter:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
```

```
#include <sys/wait.h>
int main(int argc, char *argv[])
{
    char command_str[128];
    int pid;
    int child_status;
    int wait_for = 1;
    while (1) {
        printf("sh> ");
        scanf("%s", command_str);
        pid = fork();
        if (pid == 0) {
            /* child */
            printf("cmd '%s'\n", command_str);
            execl(command_str, command_str, (char *)NULL);
            /* We should not return from execl, so only get
            to this line if it failed */
            perror("exec");
            exit(1);
        }
        if (wait_for) {
            waitpid(pid, &child_status, 0);
            printf("Done, status %d\n", child_status);
        }
    }
    return 0;
}
```

It might seem odd to have one function that duplicates an existing process and another that discards its resources and loads a different program into memory, especially since it is common for a fork to be followed almost immediately by exec. Most operating systems combine the two actions into a single call.

There are distinct advantages, however. For example, it makes it very easy to implement redirection and pipes in the shell. Imagine that you want to get a directory listing, this is the sequence of events:

1. You type `ls` at the shell prompt.
2. The shell forks a copy of itself.
3. The child execs `/bin/ls`.
4. The `ls` program prints the directory listing to `stdout` (file descriptor 1) which is attached to the terminal. You see the directory listing.
5. The `ls` program terminates and the shell regains control.

Now, imagine that you want the directory listing to be written to a file by redirecting the output using the `>` character. The sequence is now as follows:

1. You type `ls > listing.txt`.
2. The shell forks a copy of itself.
3. The child opens and truncates the file `listing.txt`, and uses `dup2(2)` to copy the file descriptor of the file over file descriptor 1 (`stdout`).
4. The child execs `/bin/ls`.
5. The program prints the listing as before, but this time it is writing to `listing.txt`.
6. The `ls` program terminates and the shell regains control.

Note that there is an opportunity at step three to modify the environment of the child process before executing the program. The `ls` program does not need to know that it is writing to a file rather than a terminal. Instead of a file, `stdout` could be connected to a pipe and so the `ls` program, still unchanged, can send output to another program. This is part of the Unix philosophy of combining many small components that each do a job well, as described in *The Art of Unix Programming*, by Eric Steven Raymond, Addison Wesley; (23 Sept. 2003) ISBN 978-0131429017, especially in the section *Pipes, Redirection, and Filters*.

Daemons

We have encountered daemons in several places already. A daemon is a process that runs in the background, owned by the `init` process, PID1, and not connected to a controlling terminal. The steps to create a daemon are as follows:

1. Call `fork()` to create a new process, after which the parent should exit, thus creating an orphan which will be re-parented to `init`.
2. The child process calls `setsid(2)`, creating a new session and process group of which it is the sole member. The exact details do not matter here, you can simply consider this as a way of isolating the process from any controlling terminal.
3. Change the working directory to the root.
4. Close all file descriptors and redirect `stdin`, `stdout`, and `stderr` (descriptors 0, 1, and 2) to `/dev/null` so that there is no input and all output is hidden.

Thankfully, all of the preceding steps can be achieved with a single function call, `daemon(3)`.

Inter-process communication

Each process is an island of memory. You can pass information from one to another in two ways. Firstly, you can copy it from one address space to the other. Secondly, you can create an area of memory that both can access and so share the data.

The first is usually combined with a queue or buffer so that there is a sequence of messages passing between processes. This implies copying the message twice: first to a holding area and then to the destination. Some examples of this are sockets, pipes, and POSIX message queues.

The second way requires not only a method of creating memory that is mapped into two (or more) address spaces at once, but also a means of synchronizing access to that memory, for example, by using semaphores or mutexes. POSIX has functions for all of these.

There is an older set of APIs known as System V IPC, which provides message queues, shared memory, and semaphores, but it is not as flexible as the POSIX equivalents so I will not describe it here. The man page on `svipc(7)` gives an overview of the facilities and there is more detail in *The Linux Programming Interface*, by Michael Kerrisk, No Starch Press and *Unix Network Programming, Volume 2*, by W. Richard Stevens.

Message-based protocols are usually easier to program and debug than shared memory, but are slow if the messages are large.

Message-based IPC

There are several options which I will summarize as follows. The attributes that differentiate between them are:

- Whether the message flow is uni- or bi-directorial.
- Whether the data flow is a byte stream, with no message boundary, or discrete messages with boundaries preserved. In the latter case, the maximum size of a message is important.
- Whether messages are tagged with a priority.

The following table summarizes these properties for FIFOs, sockets, and message queues:

Property	FIFO	Unix socket: stream	Unix socket: datagram	POSIX message queue
Message boundary	Byte stream	Byte stream	Discrete	Discrete

Property	FIFO	Unix socket: stream	Unix socket: datagram	POSIX message queue
Uni/bi-directional	Uni	Bi	Uni	Uni
Max message size	Unlimited	Unlimited	In the range 100 KiB to 250 KiB	Default: 8 KiB, absolute maximum: 1 MiB
Priority levels	None	None	None	0 to 32767

Unix (or local) sockets

Unix sockets fulfill most requirements and, coupled with the familiarity of the sockets API, they are by far the most common mechanism.

Unix sockets are created with the address family `AF_UNIX` and bound to a path name. Access to the socket is determined by the access permission of the socket file. As with Internet sockets, the socket type can be `SOCK_STREAM` or `SOCK_DGRAM`, the former giving a bi-directional byte stream, and the latter providing discrete messages with preserved boundaries. Unix socket datagrams are reliable, meaning that they will not be dropped or reordered. The maximum size for a datagram is system-dependent and is available via `/proc/sys/net/core/wmem_max`. It is typically 100 KiB or more.

Unix sockets do not have a mechanism for indicating the priority of a message.

FIFOs and named pipes

FIFO and named pipe are just different terms for the same thing. They are an extension of the anonymous pipe that is used to communicate between parent and child and are used to implement piping in the shell.

A FIFO is a special sort of file, created by the command `mkfifo(1)`. As with Unix sockets, the file access permissions determine who can read and write. They are uni-directional, meaning that there is one reader and usually one writer, though there may be several. The data is a pure byte stream but with a guarantee of atomicity of messages that are smaller than the buffer associated with the pipe. In other words, writes less than this size will not be split into several smaller writes and so the reader will read the whole message in one go, so long as the size of the buffer at the reader end is large enough. The default size of the FIFO buffer is 64 KiB on modern kernels and can be increased using `fcntl(2)` with `F_SETPIPE_SZ` up to the value in `/proc/sys/fs/pipe-max-size`, typically 1 MiB.

There is no concept of priority.

POSIX message queues

Message queues are identified by a name, which must begin with a forward slash / and contain only one / character: message queues are actually kept in a pseudo filesystem of the type `mqueue`. You create a queue and get a reference to an existing queue through `mq_open(3)`, which returns a file. Each message has a priority and messages are read from the queue in priority and then age order. Messages can be up to `/proc/sys/kernel/msgmax` bytes long. The default value is 8 KiB, but you can set it to be any size in the range 128 bytes to 1 MiB by writing the value to `/proc/sys/kernel/msgmax` bytes. Each message has a priority. They are read from the queue in priority then age order. Since the reference is a file descriptor, you can use `select(2)`, `poll(2)`, and other similar functions to wait for activity on the queue.

See the Linux man page `mq_overview(7)`.

Summary of message-based IPC

Unix sockets are the most often used because they offer all that is needed, except perhaps message priority. They are implemented on most operating systems, and so they confer maximum portability.

FIFOs are less used, mostly because they lack an equivalent to a datagram. On the other hand, the API is very simple, being the normal `open(2)`, `close(2)`, `read(2)`, and `write(2)` file calls.

Message queues are the least commonly used of this group. The code paths in the kernel are not optimized in the way that socket (network) and FIFO (filesystem) calls are.

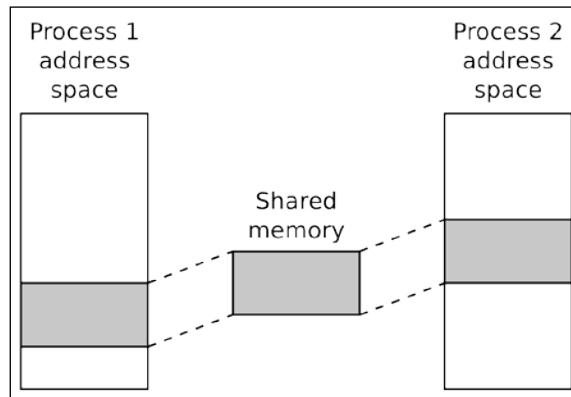
There are also higher level abstractions, in particular `dbus`, which are moving from mainstream Linux into embedded devices. `Dbus` uses Unix sockets and shared memory under the surface.

Shared memory-based IPC

Sharing memory removes the need for copying data between address spaces but introduces the problem of synchronizing accesses to it. Synchronization between processes is commonly achieved using semaphores.

POSIX shared memory

To share memory between processes, you first have to create a new area of memory and then map it into the address space of each process that wants access to it, as in the following diagram:



POSIX shared memory follows the pattern we encountered with message queues. The segments are identified by names that begin with a / character and have exactly one such character. The function `shm_open(3)` takes the name and returns a file descriptor for it. If it does not exist already and the `O_CREAT` flag is set, then a new segment is created. Initially it has a size of zero. Use the (misleadingly named) `ftruncate(2)` to expand it to the desired size.

Once you have a descriptor for the shared memory, you map it into the address space of the process using `mmap(2)`, and so threads in different processes can access the memory.

Here is an example:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/stat.h> /* For mode constants */
#include <fcntl.h>
#include <sys/types.h>
#include <errno.h>
#include <semaphore.h>
#define SHM_SEGMENT_SIZE 65536
#define SHM_SEGMENT_NAME "/demo-shm"
```

```
#define SEMA_NAME "/demo-sem"

static sem_t *demo_sem;
/*
 * If the shared memory segment does not exist already, create it
 * Returns a pointer to the segment or NULL if there is an error
 */

static void *get_shared_memory(void)
{
    int shm_fd;
    struct shared_data *shm_p;
    /* Attempt to create the shared memory segment */
    shm_fd = shm_open(SHM_SEGMENT_NAME, O_CREAT | O_EXCL | O_RDWR,
                      0666);

    if (shm_fd > 0) {
        /* succeeded: expand it to the desired size (Note: dont't do
         "this every time because ftruncate fills it with zeros) */
        printf ("Creating shared memory and setting size=%d\n",
                SHM_SEGMENT_SIZE);

        if (ftruncate(shm_fd, SHM_SEGMENT_SIZE) < 0) {
            perror("ftruncate");
            exit(1);
        }
        /* Create a semaphore as well */
        demo_sem = sem_open(SEMA_NAME, O_RDWR | O_CREAT, 0666, 1);

        if (demo_sem == SEM_FAILED)
            perror("sem_open failed\n");
    }
    else if (shm_fd == -1 && errno == EEXIST) {
        /* Already exists: open again without O_CREAT */
        shm_fd = shm_open(SHM_SEGMENT_NAME, O_RDWR, 0);
        demo_sem = sem_open(SEMA_NAME, O_RDWR);

        if (demo_sem == SEM_FAILED)
            perror("sem_open failed\n");
    }

    if (shm_fd == -1) {
        perror("shm_open " SHM_SEGMENT_NAME);
        exit(1);
    }
}
```

```
        }
        /* Map the shared memory */
        shm_p = mmap(NULL, SHM_SEGMENT_SIZE, PROT_READ | PROT_WRITE,
                     MAP_SHARED, shm_fd, 0);

        if (shm_p == NULL) {
            perror("mmap");
            exit(1);
        }
        return shm_p;
    }
int main(int argc, char *argv[])
{
    char *shm_p;
    printf("%s PID=%d\n", argv[0], getpid());
    shm_p = get_shared_memory();

    while (1) {
        printf("Press enter to see the current contents of shm\n");
        getchar();
        sem_wait(demo_sem);
        printf("%s\n", shm_p);
        /* Write our signature to the shared memory */
        sprintf(shm_p, "Hello from process %d\n", getpid());
        sem_post(demo_sem);
    }
    return 0;
}
```

The memory in Linux is taken from a `tmpfs` filesystem mounted in `/dev/shm` or `/run/shm`.

Threads

Now it is time to look at multi-threaded processes. The programming interface for threads is the POSIX threads API, which was first defined in IEEE POSIX 1003.1c standard (1995), commonly known as Pthreads. It was implemented as an additional part of the C library, `libpthread.so`. There have been two versions of Pthreads over the last 15 years or so, Linux Threads and the **Native POSIX Thread Library (NPTL)**. The latter is much more compliant with the specification, particularly with regard to the handling of signals and process IDs. It is pretty dominant now, but you may come across some older versions of uClibc that use Linux Threads.

Creating a new thread

The function to create a thread is `pthread_create(3)`:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                  void *(*start_routine) (void *), void *arg);
```

It creates a new thread of execution which begins at the function `start_routine` and places a descriptor in `pthread_t` pointed to by `thread`. It inherits the scheduling parameters of the calling thread but these can be overridden by passing a pointer to the thread attributes in `attr`. The thread will begin to execute immediately.

`pthread_t` is the main way to refer to the thread within the program but the thread can also be seen from outside using a command like `ps -elf`:

UID	PID	PPID	LWP	C	NLWP	STIME	TTY	TIME	CMD
...									
chris	6072	5648	6072	0	3	21:18	pts/0	00:00:00	./thread-demo
chris	6072	5648	6073	0	3	21:18	pts/0	00:00:00	./thread-demo

The program `thread-demo` has two threads. The `PID` and `PPID` columns show that they all belong to the same process and have the same parent, as you would expect. The column marked `LWP` is interesting, though. `LWP` stands for Light Weight Process which, in this context, is another name for thread. The numbers in that column are also known as **Thread IDs** or **TIDs**. In the main thread, the TID is the same as the PID, but for the others it is a different (higher) value. Some functions will accept a TID in places where the documentation states that you must give a PID, but be aware that this behavior is specific to Linux and not portable. Here is the code for `thread-demo`:

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/syscall.h>

static void *thread_fn(void *arg)
{
    printf("New thread started, PID %d TID %d\n",
           getpid(), (pid_t)syscall(SYS_gettid));
    sleep(10);
    printf("New thread terminating\n");
    return NULL;
}
```

```
int main(int argc, char *argv[])
{
    pthread_t t;
    printf("Main thread, PID %d TID %d\n",
        getpid(), (pid_t)syscall(SYS_gettid));
    pthread_create(&t, NULL, thread_fn, NULL);
    pthread_join(t, NULL);
    return 0;
}
```

There is a man page for `gettid(2)` which explains that you have to make the Linux `syscall` directly because there isn't a C library wrapper for it, as shown.

There is a limit to the total number of threads that a given kernel can schedule. The limit scales according to the size of the system from around 1,000 on small devices up to tens of thousands on larger embedded devices. The actual number is available in `/proc/sys/kernel/threads-max`. Once you reach this limit, `fork()` and `pthread_create()` will fail.

Terminating a thread

A thread terminates when:

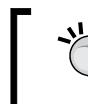
- It reaches the end of its `start_routine`
- It calls `pthread_exit(3)`
- It is canceled by another thread calling `pthread_cancel(3)`
- The process which contains the thread terminates, for example, because of a thread calling `exit(3)`, or the process receiving a signal that is not handled, masked or ignored

Note that, if a multi threaded program calls `fork(2)`, only the thread that made the call will exist in the new child process. Fork does not replicate all threads.

A thread has a return value, which is a void pointer. One thread can wait for another to terminate and collect its return value by calling `pthread_join(2)`. There is an example in the code for `thread-demo` mentioned in the preceding section. This produces a problem that is very similar to the zombie problem among processes: the resources of the thread, for example, the stack, cannot be freed up until another thread has joined with it. If threads remain unjoined there is a resource leak in the program.

Compiling a program with threads

The support for POSIX threads is part of the C library, in the library `libpthread.so`. However, there is more to building programs with threads than linking the library: there have to be changes to the way the compiler generates code to make sure that certain global variables, such as `errno`, have one instance per thread rather than one for the whole process.



When building a threaded program, you must add the switch `-pthread` at the compile and link stages.

Inter-thread communication

The big advantage of threads is that they share the address space and so can share memory variables. This is also a big disadvantage because it requires synchronization to preserve data consistency, in a similar way to memory segments shared between processes but with the proviso that, with threads, all memory is shared. Threads can create private memory using **thread local storage (TLS)**.

The `pthreads` interface provides the basics necessary to achieve synchronization: mutexes and condition variables. If you want more complex structures, you will have to build them yourself.

It is worth noting that all of the IPC methods described earlier work equally well between threads in the same process.

Mutual exclusion

To write robust programs, you need to protect each shared resource with a mutex lock and make sure that every code path that reads or writes the resource has locked the mutex first. If you apply this rule consistently, most of the problems should be solved. The ones that remain are associated with the fundamental behavior of mutexes. I will list them briefly here, but will not go into detail:

- **Deadlock:** This occurs when mutexes become permanently locked. A classic situation is the deadly embrace in which two threads each require two mutexes and have managed to lock one of them but not the other. Each block waits for the lock the other has and so they remain as they are. One simple rule which avoids the deadly embrace problem is to make sure that mutexes are always locked in the same order. Other solutions involve timeouts and back off periods.

- **Priority inversion:** The delays caused by waiting for a mutex can cause a real-time thread to miss deadlines. The specific case of priority inversion happens when a high priority thread becomes blocked waiting for a mutex locked by a low priority thread. If the low priority thread is preempted by other threads of intermediate priority, the high priority thread is forced to wait for an unbounded length of time. There are mutex protocols called priority inheritance and priority ceiling which resolve the problem at the expense of greater processing overhead in the kernel for each lock and unlock call.
- **Poor performance:** Mutexes introduce minimal overhead to code as long as threads don't have to block on them most of the time. If your design has a resource that is needed by a lot of threads, however, the contention ratio becomes significant. This is usually a design issue which can be resolved by using finer grained locking or a different algorithm.

Changing conditions

Cooperating threads need a method of alerting one another that something has changed and needs attention. That thing is called a condition and the alert is sent through a condition variable, `condvar`.

A condition is just something that you can test to give a `true` or `false` result. A simple example is a buffer that contains either zero or some items. One thread takes items from the buffer and sleeps when it is empty. Another thread places items into the buffer and signals the other thread that it has done so, because the condition that the other thread is waiting on has changed. If it is sleeping, it needs to wake up and do something. The only complexity is that the condition is, by definition, a shared resource and so has to be protected by a mutex. Here is a simple example which follows the producer-consumer relationship described in the preceding section:

```
pthread_cond_t cv = PTHREAD_COND_INITIALIZER;
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

void *consumer(void *arg)
{
    while (1) {
        pthread_mutex_lock(&mtx);
        while (buffer_empty(data))
            pthread_cond_wait(&cv, &mtx);
        /* Got data: take from buffer */
        pthread_mutex_unlock(&mtx);
        /* Process data item */
    }
}
```

```
    return NULL;
}

void *producer(void *arg)
{
    while (1) {
        /* Produce an item of data */
        pthread_mutex_lock(&mutx);
        add_data(data);
        pthread_mutex_unlock(&mutx);
        pthread_cond_signal(&cv);
    }
    return NULL;
}
```

Note that, when the consumer thread blocks on the condvar, it does so while holding a locked mutex, which would seem to be a recipe for deadlock the next time the producer thread tries to update the condition. To avoid this, `pthread_condwait(3)` unlocks the mutex after the thread is blocked and locks it again before waking it and returning from the wait.

Partitioning the problem

Now that we have covered the basics of processes and threads and the ways in which they communicate, it is time to see what we can do with them.

Here are some of the rules I use when building systems:

- **Rule 1:** Keep tasks that have a lot of interaction.
Minimize overheads by keeping closely inter-operating threads together in one process.
- **Rule 2:** Don't put all your threads in one basket.
On the other hand, try and keep components with limited interaction in separate processes, in the interests of resilience and modularity.
- **Rule 3:** Don't mix critical and non-critical threads in the same process.
This is an amplification of Rule 2: the critical part of the system, which might be the machine control program, should be kept as simple as possible and written in a more rigorous way than other parts. It must be able to continue even if other processes fail. If you have real-time threads, they, by definition, must be critical and should go into a process by themselves.

- **Rule 4:** Threads shouldn't get too intimate.

One of the temptations when writing a multi-threaded program is to intermingle the code and variables between threads because it is all in one program and easy to do. Don't keep threads modular with well-defined interactions.

- **Rule 5:** Don't think that threads are for free.

It is very easy to create additional threads but there is a cost, not least in the additional synchronization necessary to coordinate their activities.

- **Rule 6:** Threads can work in parallel.

Threads can run simultaneously on a multi-core processor, giving higher throughput. If you have a large computing job, you can create one thread per core and make maximum use of the hardware. There are libraries to help you do this, such as OpenMP. You probably shouldn't be coding parallel programming algorithms from scratch.

The Android design is a good illustration. Each application is a separate Linux process which helps to modularize memory management but especially ensures that one app crashing does not affect the whole system. The process model is also used for access control: a process can only access the files and resources which its UID and GIDs allow it to. There are a group of threads in each process. There is one to manage and update the user interface, one for handling signals from the operating system, several for managing dynamic memory allocation and the freeing up of Java objects and a worker pool of at least two threads for receiving messages from other parts of the system using the Binder protocol.

To summarize, processes provide resilience because each process has a protected memory space and, when the process terminates, all resources including memory and file descriptors are freed up, reducing resource leaks. On the other hand, threads share resources and so can communicate easily through shared variables, and can cooperate by sharing access to files and other resources. Threads give parallelism through worker pools and other abstractions which is useful on multi-core processors.

Scheduling

The second big topic I want to cover in this chapter is scheduling. The Linux scheduler has a queue of threads that are ready to run and its job is to schedule them on CPUs as they become available. Each thread has a scheduling policy which may be timeshared or real-time. The timeshared threads have a niceness value which increases or reduces their entitlement to CPU time. The real-time threads have a priority such that a higher priority thread will preempt a lower one. The scheduler works with threads, not processes. Each thread is scheduled regardless of which process it is running in.

The scheduler runs when:

- A thread blocks by calling `sleep()` or in a blocking I/O call
- A timeshare thread exhausts its time slice
- An interrupt causes a thread to be unblocked, for example, because of I/O completing

For background information on the Linux scheduler, I recommend reading the chapter on process scheduling in *Linux Kernel Development, 3rd edition by Robert Love, Addison-Wesley Professional; (July 2, 2010) ISBN-10: 0672329468*.

Fairness versus determinism

I have grouped the scheduling policies into categories of timeshare and real-time. Timeshare policies are based on the principle of fairness. They are designed to make sure that each thread gets a fair amount of processor time and that no thread can hog the system. If a thread runs for too long it is put to the back of the queue so that others can have a go. At the same time, a fairness policy needs to adjust to threads that are doing a lot of work and give them the resources to get the job done. Timeshare scheduling is good because of the way it automatically adjusts to a wide range of workloads.

On the other hand, if you have a real-time program, fairness is not helpful. Instead, you then want a policy that is deterministic, that will give you at least minimal guarantees that your real-time threads will be scheduled at the right time so that they don't miss their deadlines. This means that a real-time thread must preempt timeshare threads. Real-time threads also have a static priority that the scheduler can use to choose between them when there are several of them to run at once. The Linux real-time scheduler implements a fairly standard algorithm which runs the highest priority real-time thread. Most RTOS schedulers are also written in this way.

Both types of thread can coexist. Those requiring deterministic scheduling are scheduled first and the time remaining is divided between the timeshare threads.

Timeshare policies

Timeshare policies are designed for fairness. From Linux 2.6.23 onwards, the scheduler used has been the **Completely Fair Scheduler (CFS)**. It does not use timeslices in the normal sense of the word. Instead, it calculates a running tally of the length of time a thread would be entitled to run if it had its fair share of CPU time, and balances that with the actual amount of time it has run. If it exceeds its entitlement, and there are other timeshare threads waiting to run, the scheduler will suspend the thread and run a waiting thread instead.

The timeshare policies are:

- `SCHED_NORMAL` (also known as `SCHED_OTHER`): This is the default policy. The vast majority of Linux threads use this policy.
- `SCHED_BATCH`: This is similar to `SCHED_NORMAL` except threads are scheduled with a larger granularity; that is they run for longer but have to wait longer until scheduled again. The intention is to reduce the number of context switches for background processing (batch jobs) and so reduce the amount of CPU cache churn.
- `SCHED_IDLE`: These threads are run only when there are no threads of any other policy ready to run. It is the lowest possible priority.

There are two pairs of functions to get and set the policy and priority of a thread. The first pair takes a PID as a parameter and affects the main thread in a process:

```
struct sched_param {  
    ...  
    int sched_priority;  
    ...  
};  
int sched_setscheduler(pid_t pid, int policy,  
                      const struct sched_param *param);  
int sched_getscheduler(pid_t pid);
```

The second pair operates on `pthread_t` and so can change the parameters of the other threads in a process:

```
pthread_setschedparam(pthread_t thread, int policy,  
                      const struct sched_param *param);  
pthread_getschedparam(pthread_t thread, int *policy,  
                      struct sched_param *param);
```

Niceness

Some timeshare threads are more important than others. You can indicate this with the `nice` value which multiplies a thread's CPU entitlement by a scaling factor. The name comes from the function call, `nice(2)`, which has been part of Unix since the early days. A thread becomes nice by reducing its load on the system, or moves in the opposite direction by increasing it. The range of values is from 19, which is really nice, to -20 which is really not nice. The default value is 0, which is averagely nice or so-so.

The `nice` value can be changed for `SCHED_NORMAL` and `SCHED_BATCH` threads. To reduce niceness, which increases the CPU load, you need the capability `CAP_SYS_NICE`, which is available to the root user.

Almost all the documentation for functions and commands that change the `nice` value (`nice(2)` and the `nice` and `renice` commands) talks in terms of processes. However, it really relates to threads. As mentioned in the preceding section, you can use a TID in place of a PID to change the `nice` value of an individual thread. One other discrepancy in the standard descriptions of `nice`: the `nice` value is referred to as the priority of a thread (or sometimes, mistakenly, a process). I believe this is misleading and confuses the concept with real-time priority which is a completely different thing.

Real-time policies

Real-time policies are intended for determinism. The real-time scheduler will always run the highest priority real-time thread that is ready to run. Real-time threads always preempt timeshare threads. In essence, by selecting a real-time policy over a timeshare policy, you are saying that you have inside knowledge of the expected scheduling of this thread and wish to override the scheduler's built-in assumptions.

There are two real-time policies:

- `SCHED_FIFO`: This is a run to completion algorithm, which means that, once the thread starts to run, it will continue until it is preempted by a higher priority real-time thread or blocks in a system call or terminates (completes).
- `SCHED_RR`: This is a round robin algorithm which will cycle between threads of the same priority if they exceed their time slice which, by default, is 100 ms. Since Linux 3.9, it has been possible to control the `timeslice` value through `/proc/sys/kernel/sched_rr_timeslice_ms`. Apart from this, it behaves in the same way as `SCHED_FIFO`.

Each real-time thread has a priority in the range 1 to 99, with 99 being the highest.

To give a thread a real-time policy, you need `CAP_SYS_NICE` which, by default, is given only to the root user.

One problem with real-time scheduling, both in Linux and elsewhere, is that of a thread that becomes compute bound, often because a bug has caused it to loop indefinitely, which prevents real-time threads of lower priority from running as well as all the timeshare threads. The system become erratic and may lock up completely. There are a couple of ways to guard against this possibility.

First, since Linux 2.6.25, the scheduler has, by default, reserved 5% of CPU time for non real-time threads, so that even a runaway real-time thread cannot completely halt the system. It is configured via two kernel controls:

- `/proc/sys/kernel/sched_rt_period_us`
- `/proc/sys/kernel/sched_rt_runtime_us`

They have default values of 1,000,000 (1 second) and 950,000 (950 ms) respectively, which means that out of every second, 50ms is reserved for non real-time processing. If you want real-time threads to be able to take 100% then set `sched_rt_runtime_us` to -1.

The second option is to use a watchdog, either hardware or software, to monitor the execution of key threads and to take action when they begin to miss deadlines.

Choosing a policy

In practice, timeshare policies satisfy the majority of computing workloads. Threads that are I/O bound spend a lot of time blocked and so always have some spare entitlement in hand. When they unblock they will be scheduled almost immediately. Meanwhile, CPU-bound threads will naturally take up any CPU cycles left over. Positive nice values can be applied to the less important threads and negative values to the important ones.

Of course, this is only average behavior, there are no guarantees that this will always be the case. If more deterministic behavior is needed, then real-time policies will be required. The things that mark out a thread as being real-time are:

- It has a deadline by which it must generate an output
- Missing the deadline would compromise the effectiveness of the system
- It is event-driven
- It is not compute bound

Examples of real-time tasks include the classic robot arm servo controller, multimedia processing, and communication processing.

Choosing a real-time priority

Choosing real-time priorities that work for all expected workloads is a tricky business and a good reason for avoiding real-time policies in the first place.

The most widely used procedure for choosing priorities is known as **Rate Monotonic Analysis (RMA)**, after the 1973 paper by Liu and Layland. It applies to real-time systems with periodic threads, which is a very important class. Each thread has a period, and a utilization, which is the proportion of the period it will be executing. The goal is to balance the load so that all threads can complete their execution phase before the next period. RMA states that this can be achieved if:

- The highest priorities are given to the threads with the shortest periods
- The total utilization is less than 69%

The total utilization is the sum of all of the individual utilizations. It also makes the assumption that the interaction between threads or the time spent blocked on mutexes and the like, is negligible.

Further reading

The following resources have further information about the topics introduced in this chapter:

- *The Art of Unix Programming*, by Eric Steven Raymond, Addison Wesley; (23 Sept. 2003) ISBN 978-0131429017
- *Linux System Programming, 2nd edition*, by Robert Love, O'Reilly Media; (8 Jun. 2013) ISBN-10: 1449339530
- *Linux Kernel Development, 3rd edition* by Robert Love, Addison-Wesley Professional; (July 2, 2010) ISBN-10: 0672329468
- *The Linux Programming Interface*, by Michael Kerrisk, No Starch Press; (October 2010) ISBN 978-1-59327-220-3
- *UNIX Network Programming: v. 2: Interprocess Communications, 2nd Edition*, by W. Richard Stevens, Prentice Hall; (25 Aug. 1998) ISBN-10: 0132974290
- *Programming with POSIX Threads*, by Butenhof, David R, Addison-Wesley, Professional
- *Scheduling Algorithm for multiprogramming in a Hard-Real-Time Environment*, by C. L. Liu and James W. Layland, *Journal of ACM*, 1973, vol 20, no 1, pp. 46-61

Summary

The long Unix heritage that is built into Linux and the accompanying C libraries provides almost everything you need to write stable and resilient embedded applications. The issue is that, for every job, there are at least two ways to achieve the end you desire.

In this chapter, I have focused on two aspects of system design: the partitioning into separate processes, each with one or more threads to get the job done, and the scheduling of those threads. I hope that I have shed some light on this, and given you the basis for further study into all of them.

In the next chapter, I will examine another important aspect of system design, memory management.

11

Managing Memory

This chapter covers issues relating to memory management, which is an important topic for any Linux system, but especially for embedded Linux where system memory is usually in limited supply. After a brief refresher on virtual memory, I will show you how to measure memory use, how to detect problems with memory allocation, including memory leaks, and what happens when you run out of memory. You will have to understand the tools that are available, from simple tools such as `free` and `top`, to complex tools such as `mtrace` and `Valgrind`.

Virtual memory basics

To recap, Linux configures the memory management unit of the CPU to present a virtual address space to a running program that begins at zero and ends at the highest address, `0xffffffff` on a 32-bit processor. That address space is divided into pages of 4 KiB (there are rare examples of systems using other page sizes).

Linux divides this virtual address space into an area for applications, called user space, and an area for the kernel, called kernel space. The split between the two is set by a kernel configuration parameter named `PAGE_OFFSET`. In a typical 32-bit embedded system, `PAGE_OFFSET` is `0xc0000000`, giving the lower three GiB to user space and the top one GiB to kernel space. The user address space is allocated per process, so that each process runs in a sandbox, separated from the others. The kernel address space is the same for all processes: there is only one kernel.

Pages in this virtual address space are mapped to physical addresses by the **memory management unit (MMU)**, which uses page tables to perform the mapping.

Each page of virtual memory may be:

- unmapped, in which access will result in a `SIGSEGV`
- mapped to a page of physical memory that is private to the process

- mapped to a page of physical memory that is shared with other processes
- mapped and shared with a `copy on write` flag set: a write is trapped in the kernel which makes a copy of the page and maps it to the process in place of the original page before allowing the write to take place
- mapped to a page of physical memory that is used by the kernel

The kernel may additionally map pages to reserved memory regions, for example, to access registers and buffer memory in device drivers.

An obvious question is, why do we do it this way instead of simply referencing physical memory directly, as typical RTOS would?

There are numerous advantages to virtual memory, some of which are described here:

- Invalid memory accesses are trapped and applications alerted by `SIGSEGV`
- Processes run in their own memory space, isolated from others
- Efficient use of memory through the sharing of common code and data, for example, in libraries
- The possibility of increasing the apparent amount of physical memory by adding swap files, although swapping on embedded targets is rare

These are powerful arguments, but we have to admit that there are some disadvantages as well. It is difficult to determine the actual memory budget of an application, which is one of the main concerns of this chapter. The default allocation strategy is to over-commit, which leads to tricky out-of-memory situations, which I will also discuss later on. Finally, the delays introduced by the memory management code in handling exceptions – page faults – make the system less deterministic, which is important for real-time programs. I will cover this in *Chapter 14, Real-time Programming*.

Memory management is different for kernel space and user space. The following sections describe the essential differences and the things you need to know.

Kernel space memory layout

Kernel memory is managed in a fairly straightforward way. It is not demand-paged, meaning that, for every allocation using `kmalloc()` or similar function, there is real physical memory. Kernel memory is never discarded or paged out.

Some architectures show a summary of the memory mapping at boot time in the kernel log messages. This trace is taken from a 32-bit ARM device (a BeagleBone Black):

```
Memory: 511MB = 511MB total
Memory: 505980k/505980k available, 18308k reserved, 0K highmem
Virtual kernel memory layout:
    vector   : 0xfffff0000 - 0xfffff1000   (   4 kB)
    fixmap   : 0xfff00000 - 0xffffe0000   ( 896 kB)
    vmalloc  : 0xe0800000 - 0xff000000   ( 488 MB)
    lowmem   : 0xc0000000 - 0xe0000000   ( 512 MB)
    pkmap    : 0xbfe00000 - 0xc0000000   (   2 MB)
    modules  : 0xbff80000 - 0xbfe00000   (   6 MB)
        .text  : 0xc0008000 - 0xc0763c90   (7536 kB)
        .init  : 0xc0764000 - 0xc079f700   ( 238 kB)
        .data  : 0xc07a0000 - 0xc0827240   ( 541 kB)
        .bss   : 0xc0827240 - 0xc089e940   ( 478 kB)
```

The figure of 505980 KiB available is the amount of free memory the kernel sees when it begins execution but before it begins making dynamic allocations.

Consumers of kernel-space memory include the following:

- The kernel itself, in other words, the code and data loaded from the kernel image file at boot time. This is shown in the preceding code in the segments `.text`, `.init`, `.data`, and `.bss`. The `.init` segment is freed once the kernel has completed initialization.
- Memory allocated through the slab allocator, which is used for kernel data structures of various kinds. This includes allocations made using `kmalloc()`. They come from the region marked `lowmem`.
- Memory allocated via `vmalloc()`, usually for larger chunks of memory than is available through `kmalloc()`. These are in the `vmalloc` area.
- Mapping for device drivers to access registers and memory belonging to various bits of hardware, which you can see by reading `/proc/iomem`. These come from the `vmalloc` area but since they are mapped to physical memory that is outside of main system memory, they do not take any real memory.
- Kernel modules, which are loaded into the area marked `modules`.
- Other low level allocations that are not tracked anywhere else.

How much memory does the kernel use?

Unfortunately, there isn't a complete answer to that question, but what follows is as close as we can get.

Firstly, you can see the memory taken up by the kernel code and data in the kernel log shown previously, or you can use the `size` command, as follows:

```
$ arm-poky-linux-gnueabi-size vmlinux
text      data      bss      dec      hex    filename
9013448   796868   8428144  18238460  1164bfc  vmlinux
```

Usually, the size is small when compared to the total amount of memory. If that is not the case, you need to look through the kernel configuration and remove those components that you don't need. There is an ongoing effort to allow small kernels to be built: search for Linux-tiny or Linux Kernel Tinification. There is a project page for the latter at <https://tiny.wiki.kernel.org/>.

You can get more information about memory usage by reading `/proc/meminfo`:

```
# cat /proc/meminfo
MemTotal:        509016 kB
MemFree:         410680 kB
Buffers:          1720 kB
Cached:          25132 kB
SwapCached:       0 kB
Active:          74880 kB
Inactive:        3224 kB
Active(anon):    51344 kB
Inactive(anon):  1372 kB
Active(file):    23536 kB
Inactive(file):  1852 kB
Unevictable:      0 kB
Mlocked:          0 kB
HighTotal:        0 kB
HighFree:          0 kB
LowTotal:        509016 kB
LowFree:         410680 kB
SwapTotal:        0 kB
SwapFree:          0 kB
Dirty:            16 kB
Writeback:        0 kB
AnonPages:       51248 kB
Mapped:          24376 kB
```

Shmem:	1452 kB
Slab:	11292 kB
SReclaimable:	5164 kB
SUnreclaim:	6128 kB
KernelStack:	1832 kB
PageTables:	1540 kB
NFS_Unstable:	0 kB
Bounce:	0 kB
WritebackTmp:	0 kB
CommitLimit:	254508 kB
Committed_AS:	734936 kB
VmallocTotal:	499712 kB
VmallocUsed:	29576 kB
VmallocChunk:	389116 kB

There is a description of each of these fields in the man page for `proc(5)`. The kernel memory usage is the sum of:

- **Slab:** The total memory allocated by the slab allocator
- **KernelStack:** The stack space used when executing kernel code
- **PageTables:** The memory used for storing page tables
- **VmallocUsed:** The memory allocated by `vmalloc()`

In the case of slab allocations, you can get more information by reading `/proc/slabinfo`. Similarly, there is a breakdown of allocations in `/proc/vmallocinfo` for the `vmalloc` area. In both cases, you need detailed knowledge of the kernel and its subsystems to see exactly which subsystem is making the allocations and why, which is beyond the scope of this discussion.

With modules, you can use `lsmod` to find out the memory space taken up by the code and data:

```
# lsmod
Module           Size  Used by
g_multi          47670  2
libcomposite     14299  1 g_multi
mt7601Usta      601404  0
```

That leaves the low level allocations of which there is no record, and which prevent us from generating an accurate account of kernel space memory usage. This will appear as missing memory when we add up all the kernel and user space allocations that we know about.

User space memory layout

Linux employs a lazy allocation strategy for user space, only mapping physical pages of memory when the program accesses it. For example, allocating a buffer of 1 MiB using `malloc(3)` returns a pointer to a block of memory addresses but no actual physical memory. A flag is set in the page table entries such that any read or write access is trapped by the kernel. This is known as a page fault. Only at this point does the kernel attempt to find a page of physical memory and add it to the page table mapping for the process. It is worthwhile demonstrating this with a simple program like this one:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#define BUFFER_SIZE (1024 * 1024)

void print_pgfaults(void)
{
    int ret;
    struct rusage usage;
    ret = getrusage(RUSAGE_SELF, &usage);
    if (ret == -1) {
        perror("getrusage");
    } else {
        printf ("Major page faults %ld\n", usage.ru_majflt);
        printf ("Minor page faults %ld\n", usage.ru_minflt);
    }
}

int main (int argc, char *argv[])
{
    unsigned char *p;
    printf("Initial state\n");
    print_pgfaults();
    p = malloc(BUFFER_SIZE);
    printf("After malloc\n");
    print_pgfaults();
    memset(p, 0x42, BUFFER_SIZE);
    printf("After memset\n");
    print_pgfaults();
    memset(p, 0x42, BUFFER_SIZE);
    printf("After 2nd memset\n");
    print_pgfaults();
    return 0;
}
```

When you run it, you will see something like this:

```
Initial state
Major page faults 0
Minor page faults 172
After malloc
Major page faults 0
Minor page faults 186
After memset
Major page faults 0
Minor page faults 442
After 2nd memset
Major page faults 0
Minor page faults 442
```

There were 172 minor page faults encountered initializing the program's environment, and a further 14 when calling `getrusage(2)` (these numbers will vary depending on the architecture and the version of the C library you are using). The important part is the increase when filling the memory with data: $442 - 186 = 256$. The buffer is 1 MiB, which is 256 pages. The second call to `memset(3)` makes no difference because all the pages are now mapped.

As you can see, a page fault is generated when the kernel traps an access to a page that has not been mapped. In fact, there are two kinds of page fault: minor and major. With a minor fault, the kernel just has to find a page of physical memory and map it into the process address space, as shown in the preceding code. A major page fault occurs when the virtual memory is mapped to a file, for example using `mmap(2)`, which I will describe shortly. Reading from this memory means that the kernel not only has to find a page of memory and map it in, but it also has to be filled with data from the file. Consequently, major faults are much more expensive in time and system resources.

Process memory map

You can see the memory map for a process through the `proc` filesystem. As an example, here is the map for the `init` process, PID 1:

```
# cat /proc/1/maps
00008000-0000e000 r-xp 00000000 00:0b 23281745    /sbin/init
00016000-00017000 rwxp 00006000 00:0b 23281745    /sbin/init
00017000-00038000 rwxp 00000000 00:00 0          [heap]
b6ded000-b6f1d000 r-xp 00000000 00:0b 23281695    /lib/libc-2.19.so
b6f1d000-b6f24000 ---p 00130000 00:0b 23281695    /lib/libc-2.19.so
```

Managing Memory

```
b6f24000-b6f26000 r-xp 0012f000 00:0b 23281695 /lib/libc-2.19.so
b6f26000-b6f27000 rwxp 00131000 00:0b 23281695 /lib/libc-2.19.so
b6f27000-b6f2a000 rwxp 00000000 00:00 0
b6f2a000-b6f49000 r-xp 00000000 00:0b 23281359 /lib/ld-2.19.so
b6f4c000-b6f4e000 rwxp 00000000 00:00 0
b6f4f000-b6f50000 r-xp 00000000 00:00 0 [sigpage]
b6f50000-b6f51000 r-xp 0001e000 00:0b 23281359 /lib/ld-2.19.so
b6f51000-b6f52000 rwxp 0001f000 00:0b 23281359 /lib/ld-2.19.so
beea1000-beec2000 rw-p 00000000 00:00 0 [stack]
fffff0000-fffff1000 r-xp 00000000 00:00 0 [vectors]
```

The first three columns show the start and end virtual addresses and the permissions for each mapping. The permissions are shown here:

- **r** = read
- **w** = write
- **x** = execute
- **s** = shared
- **p** = private (copy on write)

If the mapping is associated with a file, the filename appears in the final column, and columns four, five, and six contain the offset from the start of the file, the block device number and the inode of the file. Most of the mappings are to the program itself and the libraries it is linked with. There are two areas where the program can allocate memory, marked **[heap]** and **[stack]**. Memory allocated using `malloc(3)` comes from the former (except for very large allocations, which we will come to later) ; allocations on the stack come from the latter. The maximum size of both areas is controlled by the process's `ulimit`:

- **heap**: `ulimit -d`, default unlimited
- **stack**: `ulimit -s`, default 8 MiB

Allocations that exceed the limit are rejected by `SIGSEGV`.

When running out of memory, the kernel may decide to discard pages that are mapped to a file and are read-only. If that page is accessed again, it will cause a major page fault and be read back in from the file.

Swap

The idea of swapping is to reserve some storage where the kernel can place pages of memory that are not mapped to a file, so that it can free up the memory for other uses. It increases the effective size of physical memory by the size of the swap file. It is not a panacea: there is a cost to copying pages to and from a swap file which becomes apparent on a system that has too little real memory for the workload it is carrying and begins *disk thrashing*.

Swap is seldom used on embedded devices because it does not work well with flash storage where constant writing would wear it out quickly. However, you may want to consider swapping to compressed RAM (zram).

Swap to compressed memory (zram)

The zram driver creates RAM-based block devices named `/dev/zram0`, `/dev/zram1`, and so on. Pages written to these devices are compressed before being stored. With compression ratios in the range of 30% to 50%, you can expect an overall increase in free memory of about 10%, at the expense of more processing and a corresponding increase in power usage. It is used in some low memory Android devices.

To enable zram, configure the kernel with these options:

```
CONFIG_SWAP  
CONFIG_CGROUP_MEM_RES_CTLR  
CONFIG_CGROUP_MEM_RES_CTLR_SWAP  
CONFIG_ZRAM
```

Then, mount zram at boot time by adding this to `/etc/fstab`:

```
/dev/zram0 none swap defaults zramsize=<size in  
bytes>,swapprio=<swap partition priority>
```

You can turn swap on and off by using these commands:

```
# swapon /dev/zram0  
# swapoff /dev/zram0
```

Mapping memory with mmap

A process begins life with a certain amount of memory mapped to the text (the code) and data segments of the program file, together with the shared libraries that it is linked with. It can allocate memory on its heap at runtime using `malloc(3)` and on the stack through locally scoped variables and memory allocated through `alloca(3)`. It may also load libraries dynamically at runtime using `dlopen(3)`. All of these mappings are taken care of by the kernel. However, a process can also manipulate its memory map in an explicit way using `mmap(2)`:

```
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```

It maps `length` bytes of memory from the file with the descriptor `fd`, starting at `offset` in the file, and returns a pointer to the mapping, assuming it is successful. Since the underlying hardware works in pages, the `length` is rounded up to the nearest whole number of pages. The protection parameter, `prot`, is a combination of read, write, and execute permissions and the `flags` parameter contains at least `MAP_SHARED` or `MAP_PRIVATE`. There are many other flags, which are described in the man page.

There are many things you can do with `mmap`. Here are a few of them.

Using mmap to allocate private memory

You can use `mmap` to allocate an area of private memory by setting the `MAP_ANONYMOUS` flag and the `fd` file descriptor to `-1`. This is similar to allocating memory from the heap using `malloc(3)` except that the memory is page-aligned and in multiples of pages. The memory is allocated in the same area as that used for libraries. In fact, that area is referred to by some as the `mmap` area for this reason.

Anonymous mappings are better for large allocations because they do not pin down the heap with chunks of memory, which would make fragmentation more likely. Interestingly, you will find that `malloc(3)` (in glibc at least) stops allocating memory from the heap for requests over 128 KiB and uses `mmap` in this way, so in most cases, just using `malloc` is the right thing to do. The system will choose the best way of satisfying the request.

Using mmap to share memory

As we saw in *Chapter 10, Learning About Processes and Threads*, POSIX shared memory requires mmap to access the memory segment. In this case, you set the MAP_SHARED flag and use the file descriptor from `shm_open()`:

```
int shm_fd;
char *shm_p;

shm_fd = shm_open("/myshm", O_CREAT | O_RDWR, 0666);
ftruncate(shm_fd, 65536);
shm_p = mmap(NULL, 65536, PROT_READ | PROT_WRITE,
MAP_SHARED, shm_fd, 0);
```

Using mmap to access device memory

As I mentioned in *Chapter 8, Introducing Device Drivers*, it is possible for a driver to allow its device node to be mmaped and so share some of the device memory with an application. The exact implementation is dependent on the driver.

One example is the Linux frame buffer, `/dev/fb0`. The interface is defined in `/usr/include/linux/fb.h`, including an `ioctl` function to get the size of the display and the bits per pixel. You can then use mmap to ask the video driver to share the frame buffer with the application and read and write pixels:

```
int f;
int fb_size;
unsigned char *fb_mem;

f = open("/dev/fb0", O_RDWR);
/* Use ioctl FBIOGET_VSCREENINFO to find the display dimensions
   and calculate fb_size */
fb_mem = mmap(0, fb_size, PROT_READ | PROT_WRITE, MAP_SHARED, f,
0);
/* read and write pixels through pointer fb_mem */
```

A second example is the streaming video interface, Video 4 Linux, version 2, or V4L2, which is defined in `/usr/include/linux/videodev2.h`. Each video device has a node named `/dev/videoN`, starting with `/dev/video0`. There is an `ioctl` function to ask the driver to allocate a number of video buffers which you can mmap into user space. Then, it is just a question of cycling the buffers and filling or emptying them with video data, depending on whether you are playing back or capturing a video stream.

How much memory does my application use?

As with kernel space, the different ways of allocating, mapping and sharing user space memory make it quite difficult to answer this seemingly simple question.

To begin with, you can ask the kernel how much memory it thinks is available, which you can do by using the `free` command. Here is a typical example of the output:

	total	used	free	shared	buffers	cached
Mem:	509016	504312	4704	0	26456	363860
-/+ buffers/cache:	113996	395020				
Swap:	0	0	0			



At first sight, this looks like a system that is almost out of memory with only 4704 KiB free out of 509,016 KiB: less than 1%. However, note that 26,456 KiB is in buffers and a whopping 363,860 KiB is in cache. Linux believes that free memory is wasted memory and so the kernel uses free memory for buffers and caches, in the knowledge that they can be shrunk when the need arises. Removing buffers and cache from the measurement gives the true free memory, which is 395,020 KiB; 77% of the total. When using `free`, the numbers on the second line marked `-/+ buffers/cache` are the important ones.

You can force the kernel to free up caches by writing a number between 1 and 3 to `/proc/sys/vm/drop_caches`:

```
# echo 3 > /proc/sys/vm/drop_caches
```

The number is actually a bit mask which determines which of the two broad types of cache you want to free: 1 for the page cache and 2 for the dentry and inode caches combined. The exact roles of those caches is not particularly important here, only that there is memory that the kernel is using but which can be reclaimed at short notice.

Per-process memory usage

There are several metrics to measure the amount of memory a process is using. I will begin with the two that are easiest to obtain – the **virtual set size (vss)** and the **resident memory size (rss)**, both of which are available in most implementations of the `ps` and `top` commands:

- **Vss:** called VSZ in the `ps` command and VIRT in `top`, is the total amount of memory mapped by a process. It is the sum of all the regions shown in `/proc/<PID>/map`. This number is of limited interest, since only part of the virtual memory is committed to physical memory at any one time.
- **Rss:** called RSS in `ps` and RES in `top`, is the sum of memory that is mapped to physical pages of memory. This gets closer to the actual memory budget of the process, but there is a problem, if you add up the Rss of all the processes, you will get an overestimate the memory in use because some pages will be shared.

Using top and ps

The versions of `top` and `ps` from BusyBox give very limited information. The examples that follow use the full version from the `procps` package.

The `ps` command shows Vss (VSZ) and Rss (RSS) with the options, `-Aly`, and a custom format which includes `vsz` and `rss`, as shown here:

```
# ps -eo pid,tid,cls,rtprio,stat,vsz,rss,comm
```

PID	TID	CLS	RTPRIO	STAT	VSZ	RSS	COMMAND
1	1	TS	-	Ss	4496	2652	systemd
...							
205	205	TS	-	Ss	4076	1296	systemd-journal
228	228	TS	-	Ss	2524	1396	udevd
581	581	TS	-	Ss	2880	1508	avahi-daemon
584	584	TS	-	Ss	2848	1512	dbus-daemon
590	590	TS	-	Ss	1332	680	acpid
594	594	TS	-	Ss	4600	1564	wpa_supplicant

Likewise, `top` shows a summary of the free memory and memory usage per process:

```
top - 21:17:52 up 10:04,  1 user,  load average: 0.00, 0.01, 0.05
Tasks:  96 total,   1 running,  95 sleeping,   0 stopped,   0
zombie
%Cpu(s):  1.7 us,  2.2 sy,  0.0 ni, 95.9 id,  0.0 wa,  0.0 hi,
0.2 si,  0.0 st
KiB Mem:   509016 total,   278524 used,   230492 free,   25572
buffers
KiB Swap:        0 total,        0 used,        0 free,   170920
cached

PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+
COMMAND
1098 debian    20    0 29076  16m 8312 S  0.0  3.2  0:01.29
wicd-client
 595 root      20    0 64920  9.8m 4048 S  0.0  2.0  0:01.09 node
 866 root      20    0 28892  9152 3660 S  0.2  1.8  0:36.38 Xorg
```

These simple commands give you a feel of the memory usage and give the first indication that you have a memory leak when you see that the Rss of a process keeps on increasing. However, they are not very accurate in the absolute measurements of memory usage.

Using smem

In 2009, Matt Mackall began looking at the problem of accounting for shared pages in process memory measurement and added two new metrics called the **unique set size** or **Uss**, and the **proportional set size** or **Pss**:

- **Uss:** This is the amount of memory that is committed to physical memory and is unique to a process; it is not shared with any other. It is the amount of memory that would be freed if the process were to terminate.
- **Pss:** This splits the accounting of shared pages that are committed to physical memory between all the processes that have them mapped. For example, if an area of library code is 12 pages long and is shared by six processes, each will accumulate two pages in Pss. Thus, if you add the Pss numbers for all processes, you will get the actual amount of memory being used by those processes. In other words, Pss is the number we have been looking for.

The information is available in `/proc/<PID>/smaps`, which contains additional information for each of the mappings shown in `/proc/<PID>/maps`. Here is one section from such a file which provides information about the mapping for the `libc` code segment:

```
b6e6d000-b6f45000 r-xp 00000000 b3:02 2444 /lib/libc-2.13.so
Size:          864 kB
Rss:           264 kB
Pss:            6 kB
Shared_Clean:  264 kB
Shared_Dirty:   0 kB
Private_Clean: 0 kB
Private_Dirty: 0 kB
Referenced:   264 kB
Anonymous:    0 kB
AnonHugePages: 0 kB
Swap:          0 kB
KernelPageSize: 4 kB
MMUPageSize:   4 kB
Locked:        0 kB
VmFlags: rd ex mr mw me
```



Note that the Rss is 264 KiB but because it is shared between many other processes, the Pss is only 6 KiB.

There is a tool named `smem` that collates the information from the `smaps` files and presents it in various ways, including as pie or bar charts. The project page for `smem` is <https://www.selenic.com/smem>. It is available as a package in most desktop distributions. However, since it is written in Python, installing it on an embedded target requires a Python environment, which may be too much trouble for just one tool. To help with this, there is a small program named `smemcap` that captures the state from `/proc` on the target and saves it to a TAR file which can be analyzed later on the host computer. It is part of BusyBox, but it can also be compiled from the `smem` source.

Running `smem` natively, as root, you will see these results:

# smem -t						
PID	User	Command	Swap	USS	PSS	RSS
610	0	/sbin/agetty -s tty00 11	0	128	149	720
1236	0	/sbin/agetty -s ttyGS0 1	0	128	149	720
609	0	/sbin/agetty tty1 38400	0	144	163	724
578	0	/usr/sbin/acpid	0	140	173	680
819	0	/usr/sbin/cron	0	188	201	704
634	103	avahi-daemon: chroot hel	0	112	205	500

980 0	/usr/sbin/udhcpcd -S /etc	0	196	205	568
...					
836 0	/usr/bin/X :0 -auth /var	0	7172	7746	9212
583 0	/usr/bin/node autorun.js	0	8772	9043	10076
1089 1000	/usr/bin/python -O /usr/	0	9600	11264	16388

53 6		0	65820	78251	146544

You can see from the last line of the output that, in this case, the total Pss is about a half of the Rss.

If you don't have or don't want to install Python on your target, you can capture the state using `smemcap`, again as root:

```
# smemcap > smem-bbb-cap.tar
```

Then, copy the TAR file to the host and read it using `smem -s`, though this time there is no need to run as root:

```
$ smem -t -S smem-bbb-cap.tar
```

The output is identical to that when running it natively.

Other tools to consider

Another way to display Pss is via `ps_mem` (https://github.com/pixelb/ps_mem), which prints much the same information but in a simpler format. It is also written in Python.

Android also has a tool named `procrank`, which can be cross compiled for embedded Linux with a few small changes. You can get the code from https://github.com/csimmonds/procrank_linux.

Identifying memory leaks

A memory leak occurs when memory is allocated but not freed when it is no longer needed. Memory leakage is by no means unique to embedded systems, but it becomes an issue partly because targets don't have much memory in the first place, and partly because they often run for long periods of time without rebooting, allowing the leaks to become a large puddle.

You will realize that there is a leak when you run `free` or `top` and see that free memory is continually going down, even if you drop caches, as shown in the preceding section. You will be able to identify the culprit (or culprits) by looking at the Uss and Rss per process.

There are several tools for identifying memory leaks in a program. I will look at two: `mtrace` and `Valgrind`.

mtrace

`mtrace` is a component of glibc that traces calls to `malloc(3)`, `free(3)`, and related functions, and identifies areas of memory not freed when the program exits. You need to call the `mtrace()` function from within the program to begin tracing and then at runtime, write a path name to the `MALLOC_TRACE` environment variable in which the trace information is written. If `MALLOC_TRACE` does not exist or of the file cannot be opened, `mtrace` hooks are not installed. While the trace information is written in ASCII, it is usual to use the `mtrace` command to view it.

Here is an example:

```
#include <mcheck.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int j;
    mtrace();
    for (j = 0; j < 2; j++)
        malloc(100); /* Never freed:a memory leak */
        calloc(16, 16); /* Never freed:a memory leak */
    exit(EXIT_SUCCESS);
}
```

Here is what you might see when running the program and looking at the trace:

```
$ export MALLOC_TRACE=mtrace.log
$ ./mtrace-example
$ mtrace mtrace-example mtrace.log

Memory not freed:
-----
Address      Size      Caller
0x00000000001479460 0x64      at /home/chris/mtrace-example.c:11
0x000000000014794d0 0x64      at /home/chris/mtrace-example.c:11
0x00000000001479540 0x100     at /home/chris/mtrace-example.c:15
```

Unfortunately, `mtrace` does not tell you about leaked memory while the program runs. It has to terminate first.

Valgrind

Valgrind is a very powerful tool for discovering memory problems including leaks, and other things besides. One advantage is that you don't have to recompile the programs and libraries that you want to check, although it does work better if they have been compiled with the `-g` option so that they include debug symbol tables. It works by running the program in an emulated environment and trapping execution at various points. This leads to the big downside of Valgrind, which is that the program runs at a fraction of normal speed which makes it less useful for testing anything with real-time constraints.

Incidentally, the name is often mispronounced: it says in the Valgrind FAQ that the *grind* is pronounced with a short *i* -- as in *grinned* (rhymes with *tinned*) rather than *grined* (rhymes with *find*). The FAQ, documentation and downloads are available at <http://valgrind.org>.

Valgrind contains several diagnostic tools:

- **memcheck**: This is the default tool, and detects memory leaks and general misuse of memory
- **cachegrind**: This calculates the processor cache hit rate
- **callgrind**: This calculates the cost of each function call
- **helgrind**: This highlights misuse of the Pthread API, potential deadlocks, and race conditions
- **DRD**: This is another Pthread analysis tool
- **massif**: This profiles usage of the heap and stack

You can select the tool you want with the `-tool` option. Valgrind runs on the major embedded platforms: ARM (Cortex A), PPC, MIPS, and x86 in 32- and 64-bit variants. It is available as a package in both the Yocto Project and Buildroot.

To find our memory leak, we need to use the default `memcheck` tool, with the option `--leakcheck=full` to print out the lines where the leak was found:

```
$ valgrind --leak-check=full ./mtrace-example
==17235== Memcheck, a memory error detector
==17235== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward
          et al.
==17235== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for
          copyright info
==17235== Command: ./mtrace-example
==17235==
```

```
==17235==  
==17235== HEAP SUMMARY:  
==17235==     in use at exit: 456 bytes in 3 blocks  
==17235== total heap usage: 3 allocs, 0 frees, 456 bytes  
     allocated  
==17235==  
==17235== 200 bytes in 2 blocks are definitely lost in loss record  
1 of 2  
==17235==      at 0x4C2AB80: malloc (in  
    /usr/lib/valgrind/vgpreload_memcheck-linux.so)  
==17235==      by 0x40005FA: main (mtrace-example.c:12)  
==17235==  
==17235== 256 bytes in 1 blocks are definitely lost in loss record  
2 of 2  
==17235==      at 0x4C2CC70: calloc (in  
    /usr/lib/valgrind/vgpreload_memcheck-linux.so)  
==17235==      by 0x400613: main (mtrace-example.c:14)  
==17235==  
==17235== LEAK SUMMARY:  
==17235==     definitely lost: 456 bytes in 3 blocks  
==17235==     indirectly lost: 0 bytes in 0 blocks  
==17235==     possibly lost: 0 bytes in 0 blocks  
==17235==     still reachable: 0 bytes in 0 blocks  
==17235==           suppressed: 0 bytes in 0 blocks  
==17235==  
==17235== For counts of detected and suppressed errors, rerun  
with: -v  
==17235== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0  
from 0)
```

Running out of memory

The standard memory allocation policy is to **over-commit**, meaning that the kernel will allow more memory to be allocated by applications than there is physical memory. Most of the time, this works fine because it is common for applications to request more memory than they really need. It also helps in the implementation of `fork(2)`: it is safe to make a copy of a large program because the pages of memory are shared with the `copy-on-write` flag set. In the majority of cases, `fork` is followed by an `exec` function call, which unshares the memory and then loads a new program.

However, there is always the possibility that a particular workload will cause a group of processes to try to cash in on the allocations they have been promised simultaneously and so demand more than there really is. This is an **out of memory** situation, or **OOM**. At this point, there is no other alternative but to kill off processes until the problem goes away. This is the job of the out of memory killer.

Before we get to that, there is a tuning parameter for kernel allocations in `/proc/sys/vm/overcommit_memory`, which you can set to:

- 0: heuristic over-commit (this is the default)
- 1: always over-commit, never check
- 2: always check, never over-commit

Option 1 is only really useful if you run programs that work with large sparse arrays and so allocate large areas of memory but write to a small proportion of them. Such programs are rare in the context of embedded systems.

Option 2, never over-commit, seems to be a good choice if you are worried about running out of memory, perhaps in a mission or safety-critical application. It will fail allocations that are greater than the commit limit, which is the size of swap space plus total memory multiplied by the over-commit ratio. The over-commit ratio is controlled by `/proc/sys/vm/overcommit_ratio` and has a default value of 50%.

As an example, suppose you have a device with 512 MB of system RAM and you set a really conservative ratio of 25%:

```
# echo 25 > /proc/sys/vm/overcommit_ratio
# grep -e MemTotal -e CommitLimit /proc/meminfo
MemTotal:      509016 kB
CommitLimit:   127252 kB
```

There is no swap so the commit limit is 25% of `MemTotal`, as expected.

There is another important variable in `/proc/meminfo`: `Committed_AS`. This is the total amount of memory that is needed to fulfill all the allocations made so far. I found the following on one system:

```
# grep -e MemTotal -e Committed_AS /proc/meminfo
MemTotal:      509016 kB
Committed_AS: 741364 kB
```

In other words, the kernel has already promised more memory than the available memory. Consequently, setting `overcommit_memory` to 2 means that all allocations fail, regardless of `overcommit_ratio`. To get to a working system, I would have to either install double the amount of RAM or severely reduce the number of running processes, of which there are about 40.

In all cases, the final defense is the OOM killer. It uses a heuristic method to calculate a badness score between 0 and 1,000 for each process and then terminates those with the highest score until there is enough free memory. You should see something like this in the kernel log:

```
[44510.490320] eatmem invoked oom-killer: gfp_mask=0x200da,  
order=0, oom_score_adj=0  
...
```

You can force an OOM event using `echo f > /proc/sysrq-trigger`.

You can influence the badness score for a process by writing an adjustment value to `/proc/<PID>/oom_score_adj`. A value of -1000 means that the badness score can never be greater than zero and so it will never be killed; a value of +1000 means that it will always be greater than 1000 and so will always be killed.

Further reading

The following resources have further information about the topics introduced in this chapter:

- *Linux Kernel Development, 3rd Edition*, by Robert Love, Addison Wesley, O'Reilly Media; (Jun. 2010) ISBN-10: 0672329468
- *Linux System Programming, 2nd Edition*, by Robert Love, O'Reilly Media; (8 Jun. 2013) ISBN-10: 1449339530
- *Understanding the Linux VM Manager* by Mel Gorman: <https://www.kernel.org/doc/gorman/pdf/understand.pdf>
- *Valgrind 3.3 - Advanced Debugging and Profiling for Gnu/Linux Applications* by J Seward, N. Nethercote, and J. Weidendorfer, Network Theory Ltd; (1 Mar. 2008) ISBN 978-0954612054

Summary

Accounting for every byte of memory used in a virtual memory system is just not possible. However, you can find a fairly accurate figure for the total amount of free memory, excluding that taken by buffers and cache, by using the `free` command. By monitoring it over a period of time and with different workloads, you should become confident that it will remain within a given limit.

When you want to tune memory usage or identify sources of unexpected allocations, there are resources that give more detailed information. For kernel space, the most useful information is in `/proc: meminfo, slabinfo, and vmallocinfo`.

When it comes to getting accurate measurements for user space, the best metric is Pss, as shown by `smem` and other tools. For memory debugging, you can get help from simple tracers such as `mtrace`, or you have the heavyweight option of the Valgrind `memcheck` tool.

If you have concerns about the consequence of an out of memory situation, you can fine-tune the allocation mechanism via `/proc/sys/vm/overcommit_memory` and you can control the likelihood of particular processes being killed though the `oom_score_adj` parameter.

The next chapter is all about debugging user space and kernel code using the GNU debugger, and the insights you can gain from watching code as it runs, including the memory management functions I have described here.

12

Debugging with GDB

Bugs happen. Identifying and fixing them is part of the development process. There are many different techniques for finding and characterizing program defects, including static and dynamic analysis, code review, tracing, profiling, and interactive debugging. I will look at tracers and profilers in the next chapter, but here I want to concentrate on the traditional approach of watching code execution through a debugger, in our case, the **GNU debugger, GDB**. GDB is a powerful and flexible tool. You can use it to debug applications, examine the postmortem files (`core` files) that are created after a program crash, and even step through kernel code.

In this chapter, I will show you how to use GDB to debug applications, how to look at core files and how to debug kernel code, in all cases, emphasizing the aspects that are relevant for embedded Linux.

The GNU debugger

GDB is a source-level debugger for the compiled languages, primarily C and C++, although there is also support for a variety of other languages such as Go and Objective-C. You should read the notes for the version of GDB you are using to find out the current status of support for the various languages. The project website is <http://www.gnu.org/software/gdb> and contains a lot of useful information, including the GDB manual.

Out of the box, GDB has a command-line user interface which some people find off-putting although, in reality, it is easy to use with a little practice. If command-line interfaces are not to your liking, there are a lot of front-end user interfaces to GDB and I will describe three of them later.

Preparing to debug

You need to compile the code you want to debug with debug symbols. GCC offers two options for this: `-g` and `-ggdb`. The latter adds debug information that is specific to GDB, whereas the former generates information in an appropriate format for whichever target operating system you are using, making it the more portable option. In our particular case, the target operating system is always Linux and it makes little difference whether you use `-g` or `-ggdb`. Of more interest is the fact that both options allow you to specify the level of debug information, from 0 to 3:

- 0: This produces no debug information at all and is equivalent to omitting the `-g` or `-ggdb` switch
- 1: This produces little information but which includes function names and external variables which is enough to generate a back trace
- 2: This is the default and includes information about local variables and line numbers so that you can do source level debugging and a single step through the code
- 3: This includes extra information which, among other things, means that GDB handles macro expansions correctly

In most cases, `-g` suffices but reserve `-g3` or `-ggdb3` if you are having problems stepping through code, especially if it contains macros.

The next issue to consider is the level of code optimization. Compiler optimization tends to destroy the relationship between lines of source code and machine code, which makes stepping through the source unpredictable. If you experience problems like this you will most likely need to compile without optimization, leaving out the `-O` compile switch, or at least reduce it to level 1, using the compile switch `-O1`.

A related issue is that of stack frame pointers, which are needed by GDB to generate a back trace of function calls up to the current one. On some architectures, GCC will not generate stack frame pointers with higher levels of optimization (`-O2`). If you find yourself in the situation that you really have to compile with `-O2` but still want back traces, you can override the default behavior with `-fno-omit-frame-pointer`. Also look out for code that has been hand optimized to leave out frame pointers through the addition of `-fomit-frame-pointer`: you may want to temporarily remove them.

Debugging applications using GDB

You can use GDB to debug applications in one of two ways. If you are developing code to run on desktops and servers, or indeed any environment where you compile and run the code on the same machine, it is natural to run GDB natively. However, most embedded development is done using a cross toolchain and hence you want to debug code running on the device, but control it from the cross-development environment where you have the source code and the tools. I will focus on the latter case since it is not so well documented and yet it is the most likely scenario for embedded developers. I am not going to describe the basics of using GDB here since there are many good references on that topic already, including the GDB manual and the suggested further reading at the end of the chapter.

I will begin with some details on working with gdbserver and then show you how to configure the Yocto Project and Buildroot for remote debug.

Remote debugging using gdbserver

The key component for remote debugging is the debug agent, gdbserver, which runs on the target and controls execution of the program being debugged. Gdbserver connects to a copy of GDB running on the host machine via a network connection or an RS-232 serial interface.

Debugging through gdbserver is almost, but not quite, the same as debugging natively. The differences are mostly centered around the fact that there are two computers involved and they have to be in the right state for debugging to take place. Here are some things to look out for:

- At the start of a debug session you need to load the program you want to debug on the target using gdbserver and then separately load GDB from your cross toolchain on the host.
- GDB and gdbserver need to connect to each other before a debug session can begin.
- GDB, running on the host, needs to be told where to look for debug symbols and source code, especially for shared libraries.
- The GDB run command does not work as expected.
- gdbserver will terminate when the debug session ends and you will need to restart it if you want another debug session.

- You need debug symbols and source code for the binaries you want to debug on the host, but not necessarily on the target. Often there is not enough storage space for them on the target and they will need to be stripped before deploying to the target.
- The GDB/gdbserver combination does not have all the features of GDB running natively: for example, gdbserver cannot follow the child after `fork()` whereas native GDB can.
- Odd things can happen if GDB and gdbserver are different versions or are the same version but configured differently. Ideally they should be built from the same source using your favorite build tool.

Debug symbols increase the size of executables dramatically, sometimes by a factor of 10. As mentioned in *Chapter 5, Building a Root Filesystem*, it can be useful to remove debug symbols without recompiling everything. The tool for the job is `strip` from your cross toolchain. You can control the aggressiveness of the `strip` with these switches:

- `--strip-all`: (default) removes all symbols
- `--strip-unneeded`: removes symbols not needed for relocation processing
- `--strip-debug`: removes only debug symbols

[ For applications and shared libraries, `--strip-all` (the default) is fine, but when it comes to kernel modules you will find that it will stop the module loading. Use `--strip-unneeded` instead. I am still working on a use case for `-strip-debug`.]

With that in mind, let's look at the specifics involved in debugging with the Yocto Project and Buildroot.

Setting up the Yocto Project

The Yocto Project builds a cross GDB for the host as part of the SDK, but you will have to make changes to your target configuration to include gdbserver in the target image. You can add the package explicitly, for example by adding this to `conf/local.conf`, noting once again that there must be a leading space at the start of this string:

```
IMAGE_INSTALL_append = " gdbserver"
```

Or, you can add `tools-debug` to `EXTRA_IMAGE_FEATURES`, which will add both `gdbserver` and `strace` to the target image (I will talk about `strace` in the next chapter):

```
EXTRA_IMAGE_FEATURES = "debug-tweaks tools-debug"
```

Setting up Buildroot

With Buildroot, you need to enable options both to build the cross GDB for the host (assuming that you are using the Buildroot internal toolchain) and to build gdbserver for the target. Specifically you need to enable:

- `BR2_PACKAGE_HOST_GDB`, in the menu **Toolchain | Build cross gdb for the host**
- `BR2_PACKAGE_GDB`, in the menu **Target packages | Debugging, profiling and benchmark | gdb**
- `BR2_PACKAGE_GDB_SERVER` in the menu **Target packages | Debugging, profiling and benchmark | gdbserver**

Starting to debug

Now that you have gdbserver installed on the target and a cross GDB on the host you can start a debug session.

Connecting GDB and gdbserver

The connection between GDB and gdbserver can be through a network or a serial interface. In the case of a network connection, you launch gdbserver with the TCP port number to listen on and, optionally, an IP address to accept connections from. In most cases you don't care which IP address is going to connect, so you can just give the port number. In this example gdbserver waits for a connection on port 10000 from any host:

```
# gdbserver :10000 ./hello-world
Process hello-world created; pid = 103
Listening on port 10000
```

Next, start the copy of GDB from your toolchain, giving the same program as an argument so that GDB can load the symbol table:

```
$ arm-poky-linux-gnueabi-gdb hello-world
```

In GDB, you use the command `target remote` to make the connection, giving the IP address or host name of the target and the port it is waiting on:

```
(gdb) target remote 192.168.1.101:10000
```

When gdbserver sees the connection from the host it prints the following:

```
Remote debugging from host 192.168.1.1
```

The procedure is similar for a serial connection. On the target, you tell gdbserver which serial port to use:

```
# gdbserver /dev/tty00 ./hello-world
```

You may need to configure the port baud rate beforehand using `stty` or a similar program. A simple example would be as follows:

```
# stty -F /dev/tty01 115200
```

There are many other options to `stty`, please read the man page for more details. It is worthwhile noting that the port must not be used for anything else, for example, you can't use a port that is being used as the system console. On the host, you make the connection to gdbserver using `target remote` plus the serial device at the host end of the cable. In most cases you will want to set the baud rate of the host serial port using the GDB command `set remotebaud`:

```
(gdb) set remotebaud 115200
(gdb) target remote /dev/ttyUSB0
```

Setting the sysroot

GDB needs to know where to find debug symbols and source code for shared libraries. When debugging natively the paths are well known and built in to GDB, but when using a cross toolchain, GDB has no way to guess where the root of the target filesystem is. You do so by setting the sysroot. The Yocto Project and Buildroot have different ways of handling library symbols so the location of the sysroot is quite different.

The Yocto Project includes debug information in the target filesystem image, so you need to unpack the target image tar file that is generated in `build/tmp/deploy/images`, for which you would need to do something like this:

```
$ mkdir ~/rootfs
$ cd ~/rootfs
$ sudo tar xf ~/poky/build/tmp/deploy/images/beaglebone/core-image-
minimal-
beaglebone.tar.bz2Then you can point sysroot to the root of the unpacked
files:
(gdb) set sysroot /home/chris/MELP/rootfs
```

Buildroot compiles libraries with minimal or full debug symbols, depending on `BR2_ENABLE_DEBUG`, puts them into the staging directory, then strips them as they are copied into target image. So, for Buildroot, the sysroot is always the staging area regardless of where the root filesystem is extracted.

GDB command files

There are some things that you need to do each time you run GDB, for example, setting the sysroot. It is convenient to put such commands into a command file and run them each time GDB is started. GDB reads commands from `$HOME/.gdbinit`, then from `.gdbinit` in the current directory and then from files specified on the command line with the `-x` parameter. However, recent versions of GDB will refuse to load `.gdbinit` from the current directory for security reasons. You can override that behavior for a single directory by adding a line like this to your `$HOME/.gdbinit`:

```
add-auto-load-safe-path /home/chris/myprog/.gdbinit
```

You can also disable the check globally by adding:

```
set auto-load safe-path /
```

My personal preference is use the `-x` parameter to point to the command file, which exposes the location of the file so I don't forget about it.

To help you set up GDB, Buildroot creates a GDB command file containing the correct sysroot command in `output/staging/usr/share/buildroot/gdbinit`. It will contain a command similar to this one:

```
set sysroot /home/chris/buildroot/output/host/usr/arm-buildroot-linux-gnueabi/sysroot
```

Overview of GDB commands

GDB has a great many commands, which are described in the online manual and in the resources mentioned in the *Further Reading* section. To help you get going as quickly as possible, here is a list of the most commonly used commands. In most cases there is a short-hand for the command, which is listed underneath the full command.

Breakpoints

The following table shows the commands for breakpoints:

Commands	Use
<code>break <location></code> <code>b <location></code>	Set a breakpoint on a function name, line number or line. Examples are: "main", "5", and "sortbug.c:42"
<code>info break</code> <code>i b</code>	List breakpoints
<code>delete break <N></code> <code>d b <N></code>	Delete breakpoint N

Running and stepping

The following table shows the commands for running and stepping:

Commands	Use
run r	Load a fresh copy of the program into memory and start it running. This does not work for remote debug using gdbserver
continue c	Continue execution from a breakpoint
Ctrl-C	Stop the program being debugged
step s	Step one line of code, stepping into any function that is called
next n	Step one line of code, stepping over a function call
finish	Run until the current function returns

Information commands

The following table shows the commands for getting information:

Commands	Use
backtrace bt	List the call stack
info threads	Display information about threads
info sharedlibrary	Display information about shared libraries
print <variable> p <variable>	Print the value of a variable, e.g. print foo
list	List lines of code around the current program counter

Running to a breakpoint

Gdbserver loads the program into memory and sets a breakpoint at the first instruction, then waits for a connection from GDB. When the connection is made you enter into a debug session. However, you will find that if you try to single step immediately you will get this message:

```
Cannot find bounds of current function
```

This is because the program is halted in code written in assembly that creates the run time environment for C and C++ programs. The first line of C or C++ code is the `main()` function. Supposing that you want to stop at `main()`, you would set a breakpoint there and then use the `continue` command (abbreviation `c`) to tell `gdbserver` to continue from the breakpoint at the start of the program and stop at `main`:

```
(gdb) break main  
Breakpoint 1, main (argc=1, argv=0xbeffffe24) at helloworld.c:8  
8 printf("Hello, world!\n");
```

If at this point you see the following:

```
warning: Could not load shared library symbols for 2 libraries,  
e.g. /lib/libc.so.6.
```

That means that you have forgotten the set `sysroot`!

This is all very different to starting a program natively, where you just type `run`. In fact, if you try typing `run` in a remote debug session, you will either see a message saying that the remote target does not support `run`, or in older versions of GDB it will just hang without any explanation.

Debugging shared libraries

To debug the libraries that are built by the build tool you will have to make a few changes to the build configuration. For libraries built outside the build environment you will have to do some extra work.

The Yocto Project

The Yocto Project builds debug variants of binary packages and puts them into `build/tmp/deploy/rpm/armv5e/libc6-dbg-2.21-r0.armv5e.rpm`. Here is an example of the debug package, for the C library in this case:

```
build/tmp/deploy/rpm/armv5e/libc6-dbg-2.21-r0.armv5e.rpm
```

You can add these debug packages selectively to your target image by adding `<package name-dbg>` to your target recipe. For `glibc`, the package is named `glibc-dbg`. Alternatively, you can simply tell the Yocto Project to install all debug packages by adding `dbg-pkgs` to `EXTRA_IMAGE_FEATURES`. Be warned that this will increase the size of the target image dramatically, perhaps by several hundred megabytes.

The Yocto Project places the debug symbols in a hidden directory named `.debug` in both the `lib` and `usr/lib`, directories. GDB knows to look for symbol information in these locations within the sysroot.

The debug packages also contain a copy of the source code which is installed into directory `usr/src/debug/<package name>` in the target image, which is one of the reasons for the increase in size. You can prevent it from happening by adding to your recipes:

```
PACKAGE_DEBUG_SPLIT_STYLE = "debug-without-src"
```

Remember, though, that when you are debugging remotely with `gdbserver`, you only need the debug symbols and source code on the host, not on the target. There is nothing to stop you from deleting the `lib/.debug`, `usr/lib/.debug` and `usr/src` directories from the copy of the image that is installed on the target.

Buildroot

Buildroot is characteristically straightforward. You just need to rebuild with line-level debug symbols, for which you need to enable the following:

- `BR2_ENABLE_DEBUG` in the menu **Build options | build packages with debugging symbols**

This will create the libraries with debug symbols in `output/host/usr/<arch>/sysroot`, but the copies in the target image are still stripped. If you need debug symbols on the target, perhaps to run GDB natively, you can disable stripping by setting **Build options | strip command for binaries on target** to `none`.

Other libraries

In addition to building with debug symbols you will have to tell GDB where to find the source code. GDB has a search path for source files, which you can see using the command `show directories`:

```
(gdb) show directories
Source directories searched: $cdir:$cwd
```

These are the default search paths: `$cdir` is the compile directory, which is the directory where the source was compiled; `$cwd` is the current working directory of GDB.

Normally these are sufficient, but if the source code has been moved you will have to use the directory command as shown here:

```
(gdb) dir /home/chris/MELP/src/lib_mylib
Source directories searched:
/home/chris/MELP/src/lib_mylib:$cdir:$cwd
```

Just-in-time debugging

Sometimes a program will start to misbehave after it has been running for a while and you would like to know what it is doing. The GDB attach feature does exactly that. I call it just-in-time debugging. It is available with both native and remote debug sessions.

In the case of remote debugging, you need to find the PID of the process to be debugged and pass it to gdbserver with the --attach option. For example, if the PID is 109 you would type:

```
# gdbserver --attach :10000 109
Attached; pid = 109
Listening on port 10000
```

That forces the process to stop as if it were at a breakpoint, allowing you to start your cross GDB in the normal way and connect to gdbserver.

When you are done you can detach, allowing the program to continue running without the debugger:

```
(gdb) detach
Detaching from program: /home/chris/MELP/helloworld/helloworld,
process 109
Ending remote debugging.
```

Debugging forks and threads

What happens when the program you are debugging forks? Does the debug session follow the parent or the child? The behavior is controlled by `follow-fork-mode` which may be `parent` or `child`, with `parent` being the default. Unfortunately, current versions of gdbserver do not support this option, so it only works for native debugging. If you really need to debug the child process while using gdbserver, a workaround is to modify the code so that the child loops on a variable immediately after the fork, giving you the opportunity to attach a new gdbserver session to it and then to set the variable so that it drops out of the loop.

When a thread in a multithreaded process hits a breakpoint, the default behavior is for all threads to halt. In most cases this is the best thing to do as it allows you to look at static variables without them being changed by the other threads. When you recommence execution of the thread, all the stopped threads start up, even if you are single stepping, and it is especially this last case that can cause problems. There is a way to modify the way GDB handles stopped threads, through a parameter called `scheduler-locking`. Normally it is `off`, but if you set it to `on`, only the thread that was stopped at the breakpoint is resumed and the others remain stopped, giving you a chance to see what the thread alone does without interference. This continues to be the case until you turn `scheduler-locking` off. Gdbserver supports this feature.

Core files

Core files capture the state of a failing program at the point that it terminates. You don't even have to be in the room with a debugger when the bug manifests itself. So when you see `Segmentation fault (core dumped)`, don't shrug; investigate the core file and extract the goldmine of information in there.

The first observation is that core files are not created by default, but only when the core file resource limit for the process is non-zero. You can change it for the current shell using `ulimit -c`. To remove all limits on the size of core files, type the following:

```
$ ulimit -c unlimited
```

By default, the core file is named `core` and is placed in the current working directory of the process, which is the one pointed to by `/proc/<PID>/cwd`. There are a number of problems with this scheme. Firstly, when looking at a device with several files named `core` it is not obvious which program generated each one. Secondly, the current working directory of the process may well be in a read-only filesystem, or there may not be enough space to store the `core` file, or the process may not have permissions to write to the current working directory.

There are two files that control the naming and placement of `core` files. The first is `/proc/sys/kernel/core_uses_pid`. Writing a 1 to it causes the PID number of the dying process to be appended to the filename, which is somewhat useful as long as you can associate the PID number with a program name from log files.

Much more useful is `/proc/sys/kernel/core_pattern`, which gives you a lot more control over `core` files. The default pattern is `core` but you can change it to a pattern composed of these meta characters:

- `%p`: the PID
- `%u`: the real UID of the dumped process

- %g: the real GID of the dumped process
- %s: number of the signal causing the dump
- %t: the time of dump, expressed as seconds since the Epoch, 1970-01-01 00:00:00 +0000 (UTC)
- %h: the hostname
- %e: the executable filename
- %E: the pathname of the executable, with slashes (/) replaced by exclamation marks (!)
- %c: the core file size soft resource limit of the dumped process

You can also use a pattern that begins with an absolute directory name so that all core files are gathered together in one place. As an example, the following pattern puts all core files into the /corefiles directory and names them with the program name and the time of the crash:

```
# echo /corefiles/core.%e.%t > /proc/sys/kernel/core_pattern
```

Following a core dump, you would find something like this:

```
$ ls /corefiles/  
core.sort-debug.1431425613
```

For more information, refer to the man page *core(5)*.

For more sophisticated processing of core files you can pipe them to a program that does some post processing. The core pattern begins with a pipe symbol | followed by the program name and parameters. My Ubuntu 14.04, for example, has this core pattern:

```
|/usr/share/apport/apport %p %s %c %P
```

Apport is the crash reporting tool used by Canonical. A crash reporting tool run in this way is run while the process is still in memory, and the kernel passes the core image data to it on standard input. Thus, this program can process the image, possibly stripping parts of it to reduce the size in the filesystem, or just scanning it at the time of the core dump for specific information. The program can look at various pieces of system data, for example, reading the /proc filesystem entries for the program, and can use ptrace system calls to operate on the program and read data from it. However, once the core image data is read from standard in, the kernel does various cleanups that make information about the process no longer available.

Using GDB to look at core files

Here is a sample GDB session looking at a core file:

```
$ arm-poky-linux-gnueabi-gdb sort-debug
/home/chris/MELP/rootdirs/rootfs/corefiles/core.sort-debug.1431425613
[...]
Core was generated by `./sort-debug'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x000085c8 in addtree (p=0x0, w=0xbeac4c60 "the") at sort-
debug.c:41
41      p->word = strdup (w);
```

That shows that the program stopped at line 43. The `list` command shows the code in the immediate vicinity:

```
(gdb) list
37      static struct tnode *addtree (struct tnode *p, char *w)
38  {
39      int cond;
40
41      p->word = strdup (w);
42      p->count = 1;
43      p->left = NULL;
44      p->right = NULL;
45
```

The `backtrace` command (shortened to `bt`) shows how we got to this point:

```
(gdb) bt
#0  0x000085c8 in addtree (p=0x0, w=0xbeac4c60 "the") at sort-
debug.c:41
#1  0x00008798 in main (argc=1, argv=0xbeac4e24) at sort-debug.c:89
```

An obvious mistake: `addtree()` was called with a null pointer.

GDB user interfaces

GDB is controlled at a low level through the GDB machine interface, GDB/MI, which is used to wrap GDB in a user interface or as part of a larger program and considerably extends the range of options available to you.

I have only mentioned those which have features that are useful in embedded development.

Terminal user interface

Terminal user interface (TUI), is an optional part of the standard GDB package. The main feature is a code window which shows the line of code about to be executed, together with any breakpoints. It is a definite improvement on the `list` command in command-line mode GDB.

The attraction of TUI is that it just works and doesn't need any extra set-up and, since it is in text mode, it is possible to use over an ssh terminal session when running `gdb` natively on a target. Most cross toolchains configure GDB with TUI. Simply add `-tui` to the command line and you will see the following:

```
sort-debug.c
36     * the count, otherwise add a new node */
37     static struct tnode *addtree (struct tnode *p, char *w)
38     {
39         int cond;
40
B+>41         p->word = strdup (w);
42         p->count = 1;
43         p->left = NULL;
44         p->right = NULL;
45
46         cond = strcmp (w, p->word);
47
48         if (cond == 0)

remote Thread 95 In: addtree                                         Line: 41   PC: 0x85b4
Breakpoint 1, main (argc=1, argv=0xbeffffe24) at sort-debug.c:72
(gdb) break addtree
Breakpoint 2 at 0x85b4: file sort-debug.c, line 41.
(gdb) c
Continuing.

Breakpoint 2, addtree (p=0x0, w=0xbefffc60 "the") at sort-debug.c:41
(gdb) █
```

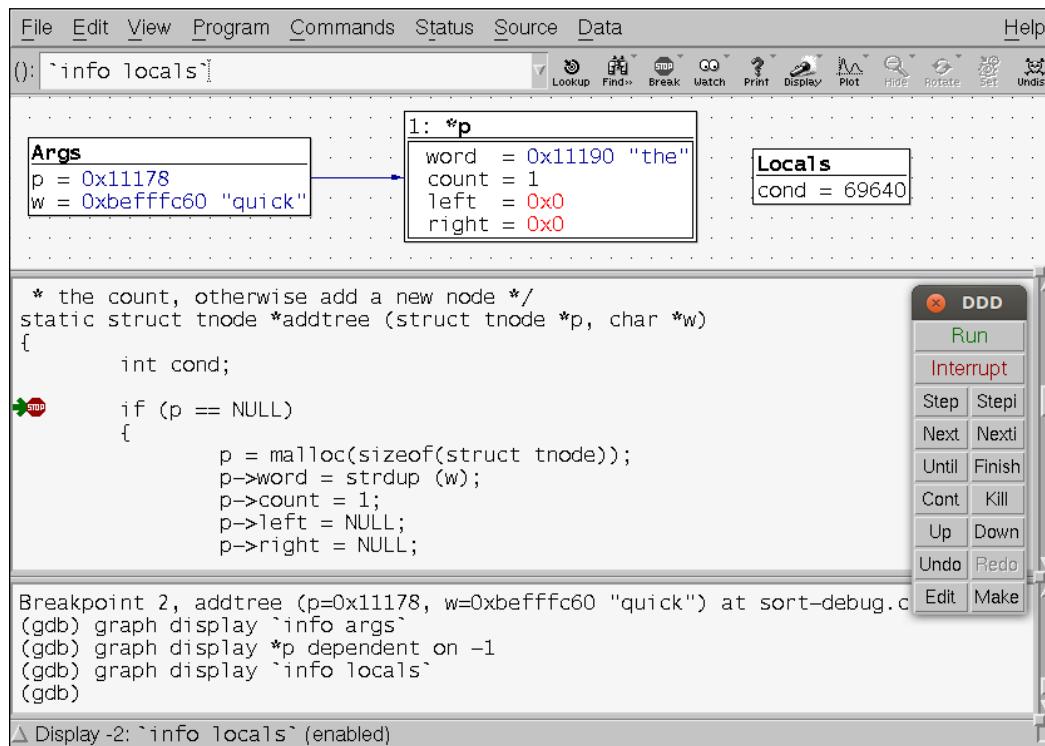
Data display debugger

Data display debugger (DDD), is a simple standalone program that gives you a graphical user interface to GDB with minimal fuss and bother and, although the UI controls look dated, it does everything that is necessary.

The `--debugger` option tells DDD to use GDB from your toolchain and you can use the `-x` argument for GDB command files:

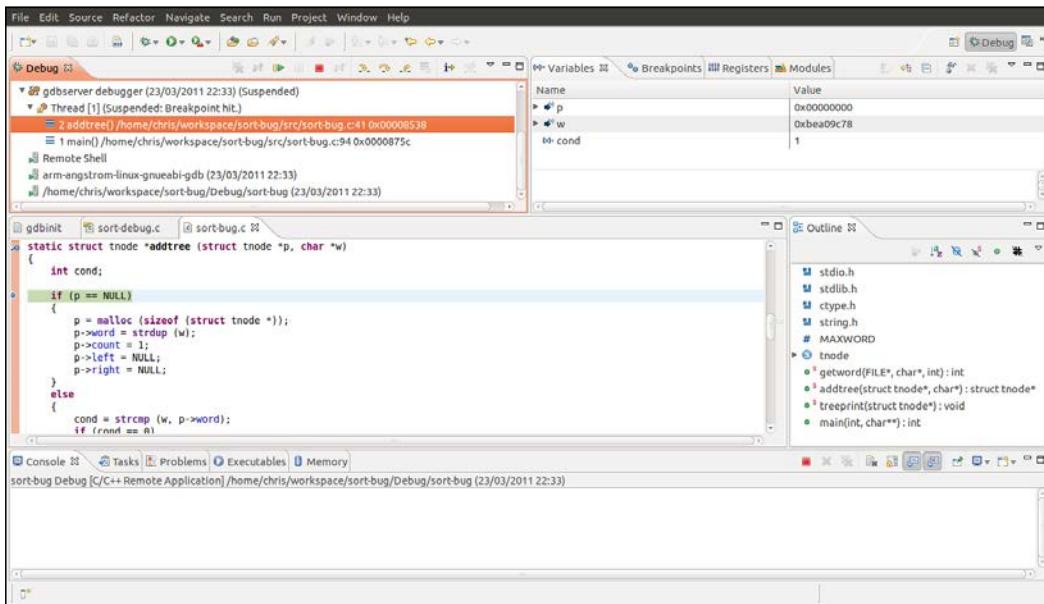
```
$ ddd --debugger arm-poky-linux-gnueabi-gdb -x gdbinit sort-debug
```

The following screenshot shows off one of the nicest features: the data window which contains items in a grid that you can rearrange as you wish. If you double-click on a pointer, it is expanded into a new data item and the link is shown with an arrow:



Eclipse

Eclipse, with the **C development toolkit (CDT)** plug-in, supports debugging with GDB, including remote debugging. If you use Eclipse for all your code development, this is the obvious tool to use but, if you are not a regular Eclipse user it is probably not worth the effort of setting it up just for this task. It would take me a whole chapter to explain adequately how to configure CDT to work with a cross toolchain and connect to a remote device, so I will refer you to the references at the end of the chapter for more information. The screenshot that follows shows the debug perspective of CDT. In the top right window you see the stack frames for each of the threads in the process, and at the top right is the watch window showing variables. In the middle is the code window, showing the line of code where the debugger has stopped the program.



Debugging kernel code

Debugging application code helps you gain insight into the way code works and what is happening when it misbehaves and you can do the same with the kernel, with some limitations.

You can use kgdb for source level debugging, in a manner similar to remote debugging with gdbserver. There is also a self-hosted kernel debugger, kdb, that is handy for lighter weight tasks such as seeing if an instruction is executed and getting the backtrace to find out how it got there. Finally, there are kernel oops messages and panics, which tell you a lot about the cause of a kernel exception.

Debugging kernel code with kgdb

When looking at kernel code using a source debugger, you must remember that the kernel is a complex system, with real-time behaviors. Don't expect debugging to be as easy as it is for applications. Stepping through code that changes the memory mapping or switches context is likely to produce odd results.

kgdb is the name given to the kernel GDB stubs that have been part of mainline Linux for many years now. There is a user manual in the kernel DocBook and you can find an online version at <https://www.kernel.org/doc/html/docs/kgdb/index.html>.

The widely supported way to connect to kgdb is over the serial interface, which is usually shared with the serial console, and so this implementation is called `kgdboc`, meaning kgdb over console. To work, it requires a platform tty driver that supports I/O polling instead of interrupts, since kgdb has to disable interrupts when communicating with GDB. A few platforms support kgdb over USB and there have been versions that work over Ethernet but, unfortunately, none of those have found their way into mainline Linux.

The same caveats about optimization and stack frames apply to the kernel, with the limitation that the kernel is written to assume an optimization level of at least `-O1`. You can override the kernel compile flags by setting `KCFLAGS` before running `make`.

These, then, are the kernel configuration options you will need for kernel debugging:

- `CONFIG_DEBUG_INFO` is in the **Kernel hacking | Compile-time checks and compiler options | Compile the kernel with debug info menu**
- `CONFIG_FRAME_POINTER` may be an option for your architecture, and is in the **Kernel hacking | Compile-time checks and compiler options | Compile the kernel with frame pointers menu**
- `CONFIG_KGDB` is in the **Kernel hacking | KGDB: kernel debugger menu**
- `CONFIG_KGDB_SERIAL_CONSOLE` is in the **Kernel hacking | KGDB: kernel debugger | KGDB: use kgdb over the serial console menu**

In addition to the `uImage` or `zImage` compressed kernel image, you will need the kernel image in ELF object format so that GDB can load the symbols into memory. That is the file called `vmlinuz` that is generated in the directory where Linux is built. In the Yocto Project, you can request that a copy be included in the target image, which is convenient for this and other debug tasks. It is built into a package named `kernel-vmlinuz`, which you can install like any other, for example by adding it to the `IMAGE_INSTALL_append` list. The file is put into the boot directory, with a name like this:

```
boot/vmlinuz-3.14.26ltsi-yocto-standard
```

In Buildroot, you will find `vmlinux` in the directory where the kernel was built, which is in `output/build/linux-<version string>/vmlinux`.

A sample debug session

The best way to show you how it works is with a simple example.

You need to tell `kgdb` which serial port to use, either through the kernel command line or at runtime via `sysfs`. For the first option, add `kgdboc=<tty>, <baud rate>` to the command line, as shown:

```
kgdboc=tty00,115200
```

For the second option, boot the device up and write the terminal name to the `/sys/module/kgdboc/parameters/kgdboc` file, as shown:

```
# echo tty00 > /sys/module/kgdboc/parameters/kgdboc
```

Note that you cannot set the baud rate in this way. If it is the same `tty` as the console then it is set already, if not use `stty` or a similar program.

Now you can start GDB on the host, selecting the `vmlinux` file that matches the running kernel:

```
$ arm-poky-linux-gnueabi-gdb ~/linux/vmlinux
```

GDB loads the symbol table from `vmlinux` and waits for further input.

Next, close any terminal emulator that is attached to the console: you are about to use it for GDB and, if both are active at the same time, some of the debug strings might get corrupted.

Now, you can return to GDB and attempt to connect to `kgdb`. However, you will find that the response you get from `target remote` at this time is unhelpful:

```
(gdb) set remotebaud 115200
(gdb) target remote /dev/ttys00
Remote debugging using /dev/ttys00
Bogus trace status reply from target: qTStatus
```

The problem is that `kgdb` is not listening for a connection at this point. You need to interrupt the kernel before you can enter into an interactive GDB session with it. Unfortunately, just typing `Ctrl + C` in GDB, as you would with an application, does not work. You have to force a trap into the kernel by launching another shell on the target, via `ssh`, for example, and writing a `g` to `/proc/sysrq-trigger` on the target board:

```
# echo g > /proc/sysrq-trigger
```

The target stops dead at this point. Now you can connect to kgdb via the serial device at the host end of the cable:

```
(gdb) set remotebaud 115200
(gdb) target remote /dev/ttyUSB0
Remote debugging using /dev/ttyUSB0
0xc009a59c in arch_kgdb_breakpoint ()
```

At last, GDB is in charge. You can set breakpoints, examine variables, look at backtraces, and so on. As an example, set a break on sys_sync, as follows:

```
(gdb) break sys_sync
Breakpoint 1 at 0xc0128a88: file fs/sync.c, line 103.
(gdb) c
Continuing.
```

Now the target comes back to life. Typing sync on the target calls sys_sync and hits the breakpoint.

```
[New Thread 87]
[Switching to Thread 87]

Breakpoint 1, sys_sync () at fs/sync.c:103
```

If you have finished the debug session and want to disable kgdboc, just set the kgdboc terminal to null:

```
# echo "" > /sys/module/kgdboc/parameters/kgdboc
```

Debugging early code

The preceding example works in cases where the code you are interested in is executed when the system is fully booted. If you need to get in early, you can tell the kernel to wait during boot by adding kgdbwait to the command line, after the kgdboc option:

```
kgdboc=tty00,115200 kgdbwait
```

Now, when you boot, you will see this on the console:

```
1.103415] console [tty00] enabled
[    1.108216] kgdb: Registered I/O driver kgdboc.
[    1.113071] kgdb: Waiting for connection from remote gdb...
```

At this point, you can close the console and connect from GDB in the usual way.

Debugging modules

Debugging kernel modules presents an additional challenge because the code is relocated at runtime and so you need to find out at what address it resides. The information is presented via sysfs. The relocation addresses for each section of the module are stored in /sys/module/<module name>/sections. Note that, since ELF sections begin with a dot, '.', they appear as hidden files and you will have to use ls -a if you want to list them. The important ones are .text, .data, and .bss.

Take as an example a module named mbx:

```
# cat /sys/module/mbx/sections/.text
0xbf000000
# cat /sys/module/mbx/sections/.data
0xbf0003e8
# cat /sys/module/mbx/sections/.bss
0xbf0005c0
```

Now you can use these numbers in GDB to load the symbol table for the module at those addresses:

```
(gdb) add-symbol-file /home/chris/mbx-driver/mbx.ko 0xbf000000 \
-s .data 0xbf0003e8 -s .bss 0xbf0005c0
add symbol table from file "/home/chris/mbx-driver/mbx.ko" at
.text_addr = 0xbf000000
.data_addr = 0xbf0003e8
.bss_addr = 0xbf0005c0
```

Everything should now work as normal: you can set breakpoints and inspect global and local variables in the module just as you can in vmlinux:

```
(gdb) break mbx_write

Breakpoint 1 at 0xbf00009c: file /home/chris/mbx-driver/mbx.c,
line 93.

(gdb) c
Continuing.
```

Then, force the device driver to call mbx_write and it will hit the breakpoint:

```
Breakpoint 1, mbx_write (file=0xde7a71c0, buffer=0xadf40 "hello\n\n",
length=6, offset=0xde73df80)
at /home/chris/mbx-driver/mbx.c:93
```

Debugging kernel code with kdb

Although kdb does not have the features of kgdb and GDB, it does have its uses and, being self-hosted, there are no external dependencies to worry about. kdb has a simple command-line interface which you can use on a serial console. You can use it to inspect memory, registers, process lists, dmesg, and even set breakpoints to stop in a certain location.

To configure kdb for access via a serial console, enable kgdb as shown previously and then enable this additional option:

- CONFIG_KGDB_KDB, which is in the **KGDB: Kernel hacking | kernel debugger | KGDB_KDB: include kdb frontend for kgdb** menu

Now, when you force the kernel to a trap, instead of entering into a GDB session, you will see the kdb shell on the console:

```
# echo g > /proc/sysrq-trigger
[    42.971126] SysRq : DEBUG

Entering kdb (current=0xdf36c080, pid 83) due to Keyboard Entry
kdb>
```

There are quite a few things you can do in the kdb shell. The help command will print all of the options. Here is an overview.

Getting information:

- ps: displays active processes
- ps A: displays all processes
- lsmod: lists modules
- dmesg: displays the kernel log buffer

Breakpoints:

- bp: sets a breakpoint
- bl: lists breakpoints
- bc: clears a breakpoint
- bt: prints a backtrace
- go: continues execution

Inspect memory and registers:

- md: displays memory
- rd: displays registers

Here is a quick example of setting a break point:

```
kdb> bp sys_sync
Instruction(i) BP #0 at 0xc01304ec (sys_sync)
    is enabled  addr at 00000000c01304ec, hardtype=0 installed=0

kdb> go
```

The kernel returns to life and the console shows the normal bash prompt. If you type sync, it hits the breakpoint and enters kdb again:

```
Entering kdb (current=0xdf388a80, pid 88) due to Breakpoint @
0xc01304ec
```

kdb is not a source debugger so you can't see the source code, or single step. However, you can display a backtrace using the bt command, which is useful to get an idea of program flow and call hierarchy.

When the kernel performs an invalid memory access or executes an illegal instruction, a kernel oops message is written to the kernel log. The most useful part of this is the backtrace, and I want to show you how to use the information there to locate the line of code that caused the fault. I will also address the problem of preserving oops messages if they cause the system to crash.

Looking at an oops

An oops message looks like this:

```
[ 56.225868] Unable to handle kernel NULL pointer dereference at
virtual address 00000400[ 56.229038] pgd = cb624000[ 56.229454]
[00000400] *pgd=6b715831, *pte=00000000, *ppte=00000000[ 56.231768]
Internal error: Oops: 817 [#1] SMP ARM[ 56.232443] Modules linked
in: mbx(O)[ 56.233556] CPU: 0 PID: 98 Comm: sh Tainted: G O 4.1.10
#1[ 56.234234] Hardware name: ARM-Versatile Express[ 56.234810]
task: cb709c80 ti: cb71a000 task.ti: cb71a000[ 56.236801] PC is at
mbx_write+0x14/0x98 [mbx][ 56.237303] LR is at __vfs_write+0x20/0xd8[
56.237559] pc : [<bf0000a0>] lr : [<c0307154>] psr: 800f0013[
56.237559] sp : cb71bef8 ip : bf00008c fp : 00000000[ 56.238183] r10:
00000000 r9 : cb71a000 r8 : c02107c4[ 56.238485] r7 : cb71bf88 r6 :
000afb98 r5 : 00000006 r4 : 00000000[ 56.238857] r3 : cb71bf88 r2 :
00000006 r1 : 000afb98 r0 : cb61d600

[ 56.239276] Flags: Nzcv IRQs on FIQs on Mode SVC_32 ISA ARM
Segment user[ 56.239685] Control: 10c5387d Table: 6b624059 DAC:
00000015[ 56.240019] Process sh (pid: 98, stack limit = 0xcb71a220)
```

PC is at `mbx_write+0x14/0x98` [`mbx`] tells you most of what you want to know: the last instruction was in the `mbx_write` function in a kernel module named `mbx`. Furthermore, it was at offset `0x14` bytes from the start of the function, which is `0x98` bytes long.

Next, take a look at the backtrace:

```
[ 56.240363] Stack: (0xcb71bef8 to 0xcb71c000) [ 56.240745] bee0:  
cb71bf88 cb61d600[ 56.241331] bf00: 00000006 c0307154 00000000 c020a308  
cb619d88  
00000301 00000000 00000042[ 56.241775] bf20: 00000000 cb61d608 cb709c80  
cb709c78 cb71bf60  
c0250a54 00000000 cb709ee0[ 56.242190] bf40: 00000003 bef4f658 00000000  
cb61d600 cb61d600  
00000006 000afb98 cb71bf88[ 56.242605] bf60: c02107c4 c030794c 00000000  
00000000 cb61d600  
cb61d600 00000006 000afb98[ 56.243025] bf80: c02107c4 c0308174 00000000  
00000000 00000000  
000ada10 00000001 000afb98[ 56.243493] bfa0: 00000004 c0210640 000ada10  
00000001 00000001  
000afb98 00000006 00000000[ 56.243952] bfc0: 000ada10 00000001 000afb98  
00000004 00000001  
00000020 000ae274 00000000[ 56.244420] bfe0: 00000000 bef4f49c 0000fcdc  
b6f1aedc 600f0010  
00000001 00000000 00000000[ 56.245653] [<bf0000a0>] (mbx_write [mbx])  
from [<c0307154>]  
(_vfs_write+0x20/0xd8)[ 56.246368] [<c0307154>]  
(_vfs_write) from [<c030794c>]  
(vfs_write+0x90/0x164)[ 56.246843] [<c030794c>] (vfs_write) from  
[<c0308174>]  
(SyS_write+0x44/0x9c)[ 56.247265] [<c0308174>] (SyS_write) from  
[<c0210640>]  
(ret_fast_syscall+0x0/0x3c)[ 56.247737] Code: e5904090 e3520b01  
23a02b01 e1a05002 (e5842400)[ 56.248372] ---[ end trace  
999c378e4df13d74 ]---
```

In this case, we don't learn much more, merely that `mbx_write` is called from the virtual filesystem code.

It would be very nice to find the line of code that relates to `mbx_write+0x14`, for which we can use `objdump -S`. We can see from `objdump -S` that `mbx_write` is at offset `0x8c` in `mbx.ko`, so that last instruction executed is at `0x8c + 0x14 = 0xa0`. Now, we just need to look at that offset and see what is there:

```
$ arm-poky-linux-gnueabi-objdump -S mbx.kostatic ssize_t mbx_write(struct file *file, const char *buffer, size_t length, loff_t * offset){ 8c:
e92d4038      push    {r3, r4, r5, lr}  struct mbx_data *m = (struct
mbx_data *)file->private_data; 90:   e5904090      ldr     r4, [r0,
#144] ; 0x90 94:   e3520b01      cmp     r2, #1024      ; 0x400 98:
23a02b01      movcs   r2, #1024      ; 0x400 if (length > MBX_LEN)
length = MBX_LEN; m->mbx_len = length; 9c:   e1a05002      mov
r5, r2 a0:   e5842400      str     r2, [r4, #1024] ; 0x400
```

This shows the instruction where it stopped. The last line of code is shown here:

```
m->mbx_len = length;
```

You can see that `m` has the type `struct mbx_data *`. Here is the place where that structure is defined:

```
#define MBX_LEN 1024 struct mbx_data {  char mbx[MBX_LEN];  int mbx_
len;};
```

So, it looks like the `m` variable is a null pointer, and that is causing the oops.

Preserving the oops

Decoding an oops is only possible if you can capture it in the first place. If the system crashes during boot before the console is enabled, or after a suspend, you won't see it. There are mechanisms to log kernel oops and messages to an MTD partition or to persistent memory, but here is a simple technique that works in many cases and needs little prior thought.

So long as the contents of memory are not corrupted during a reset (and usually they are not), you can reboot into the bootloader and use it to display memory. You need to know the location of the kernel log buffer, remembering that it is a simple ring buffer of text messages. The symbol is `__log_buf`. Look this up in `System.map` for the kernel:

```
$ grep __log_buf System.mapc0f72428 b __log_buf
```

Then, map that kernel logical address into a physical address that U-Boot can understand by subtracting `PAGE_OFFSET`, `0xc0000000`, and adding the physical start of RAM, `0x80000000` on a BeagleBone, so `c0f72428 - 0xc0000000 + 0x80000000 = 80f72428`.

Then use the U-Boot `md` command to show the log:

```
U-Boot# md 80f7242880f72428: 00000000 00000000 00210034
c6000000      .....4.!.....80f72438: 746f6f42 20676e69 756e694c
6e6f2078      Booting Linux on80f72448: 79687020 61636973 5043206c
78302055      physical CPU 0x80f72458: 00000030 00000000 00000000 00730084
0.....s.80f72468: a6000000 756e694c 65762078 6f697372 ....
Linux versio80f72478: 2e34206e 30312e31 68632820 40736972    n 4.1.10
(chris@80f72488: 6c697562 29726564 63672820 65762063    builder)
(gcc ve80f72498: 6f697372 2e34206e 20312e39 6f726328    rsion 4.9.1
(cro80f724a8: 6f747373 4e2d6c6f 2e312047 302e3032    ssstool-NG
1.20.080f724b8: 20292029 53203123 5720504d 4f206465    ) ) #1 SMP Wed
080f724c8: 32207463 37312038 3a31353a 47203335    ct 28 17:51:53 G
```



From Linux 3.5 onwards, there is a 16-byte binary header for each line in the kernel log buffer which encodes a timestamp, a log level and other things. There is a discussion about it in the Linux Weekly News titled *Toward more reliable logging* at <https://lwn.net/Articles/492125/>.

Additional reading

The following resources have further information about the topics introduced in this chapter:

- *The Art of Debugging with GDB, DDD, and Eclipse*, by Norman Matloff and Peter Jay Salzman, No Starch Press; 1 edition (28 Sept. 2008), ISBN 978-1593271749
- *GDB Pocket Reference* by Arnold Robbins, O'Reilly Media; 1st edition (12 May 2005), ISBN 978-0596100278
- *Getting to grips with Eclipse: cross compiling*, <http://2net.co.uk/tutorial/eclipse-cross-compile>
- *Getting to grips with Eclipse: remote access and debugging*, <http://2net.co.uk/tutorial/eclipse-rse>

Summary

GDB for interactive debugging is a useful tool in the embedded developer's tool chest. It is a stable, well-documented and well-known entity. It has the ability to debug remotely by placing an agent on the target, be it `gdbserver` for applications or `kgdb` for kernel code and, although the default command-line user interface takes a while to get used to, there are many alternative front-ends. The three I mentioned were TUI, DDD, and Eclipse, which should cover most situations, but there are other front-ends around that you can try.

A second and equally important way to approach debugging is to collect crash reports and analyze them offline. In this category, I have looked at application core dumps and kernel oops messages.

However, this is only one way of identifying flaws in programs. In the next chapter, I will talk about profiling and tracing as ways of analyzing and optimizing programs.

13

Profiling and Tracing

Interactive debugging using a source level debugger, as described in the previous chapter, can give you an insight into the way a program works, but it constrains your view to a small body of code. In this chapter, I will look at the larger picture to see if the system is performing as intended.

Programmers and system designers are notoriously bad at guessing where bottlenecks are. So, if your system has performance issues, it is wise to start by looking at the full system and then work down, using more sophisticated tools. In this chapter I begin with the well-known command, `top`, as a means of getting an overview. Often the problem can be localized to a single program, which you can analyze using the Linux profiler, `perf`. If the problem is not so localized and you want to get a broader picture, `perf` can do that as well. To diagnose problems associated with the kernel, I will describe the trace tools, `Ftrace` and `LTTng`, as a means of gathering detailed information.

I will also cover Valgrind which, because of its sandboxed execution environment, can monitor a program and report on code as it runs. I will complete the chapter with a description of a simple trace tool, `strace`, which reveals the execution of a program by tracing the system calls it makes.

The observer effect

Before diving into the tools, let's talk about what the tools will show you. As is the case in many fields, measuring a certain property affects the observation itself. Measuring the electric current in a line requires measuring the voltage drop over a small resistor. However, the resistor itself affects the current. The same is true for profiling: every system observation has a cost in CPU cycles and that resource is no longer spent on the application. Measurement tools also mess up caching behavior, eat memory space, and write to disk, which all make it worse. There is no measurement without overhead.

I've often heard engineers say that the results of a profiling job were totally misleading. That is usually because they were performing the measurements on something not approaching a real situation. Always try to measure on the target, using release builds of the software, with a valid data set, using as few extra services as possible.

Symbol tables and compile flags

We will hit a problem immediately. While it is important to observe the system in its natural state, the tools often need additional information to make sense of the events.

Some tools require special kernel options, specifically from those listed in the introduction, `perf`, `Ftrace`, and `LTTng`. Therefore, you will probably have to build and deploy a new kernel for these tests.

Debug symbols are very helpful in translating raw program addresses into function names and lines of code. Deploying executables with debug symbols does not change the execution of the code but it does require that you have copies of the binaries and the kernel compiled with debug, at least for the components you want to profile. Some tools work best if you have these installed on the target system, `perf`, for example. The techniques are the same as for general debugging, as I discussed in *Chapter 12, Debugging with GDB*.

If you want a tool to generate call graphs, you may have to compile with stack frames enabled. If you want the tool to attribute addresses with lines of code accurately, you may need to compile with lower levels of optimization.

Finally, some tools require instrumentation to be inserted into the program to capture samples, so you will have to recompile those components. This applies to `gprof` for applications, and `Ftrace` and `LTTng` for the kernel.

Be aware that, the more you change the system you are observing, the harder it is to relate the measurements you make to the production system.



It is best to adopt a wait-and-see approach, making changes only when the need is clear, and being mindful that each time you do so, you will change what you are measuring.



Beginning to profile

When looking at the entire system, a good place to start is with a simple tool like `top`, which gives you an overview very quickly. It shows you how much memory is being used, which processes are eating CPU cycles, and how this is spread across different cores and time.

If `top` shows that a single application is using up all the CPU cycles in user space then you can profile that application using `perf`.

If two or more processes have a high CPU usage, there is probably something that is coupling them together, perhaps data communication. If a lot of cycles are spent in system calls or handling interrupts, then there may be an issue with the kernel configuration or with a device driver. In either case you need to start by taking a profile of the whole system, again using `perf`.

If you want to find out more about the kernel and the sequencing of events there, you would use `Ftrace` or `LTTng`.

There could be other problems that `top` will not help you with. If you have multi-threaded code and there are problems with lockups, or if you have random data corruption then Valgrind plus the Helgrind plug-in might be helpful. Memory leaks also fit into this category: I covered memory-related diagnosis in *Chapter 11, Managing Memory*.

Profiling with top

`top` is a simple tool that doesn't require any special kernel options or symbol tables. There is a basic version in BusyBox, and a more functional version in the `procps` package which is available in the Yocto Project and Buildroot. You may also want to consider using `htop` which is functionally similar to `top` but has a nicer user interface (some people think).

To begin with, focus on the summary line of `top`, which is the second line if you are using BusyBox and the third line if using `procps top`. Here is an example, using BusyBox `top`:

```
Mem: 57044K used, 446172K free, 40K shrd, 3352K buff, 34452K cached
CPU: 58% usr 4% sys 0% nic 0% idle 37% io 0% irq 0% sirq
Load average: 0.24 0.06 0.02 2/51 105
 PID  PPID USER      STAT  VSZ %VSZ %CPU COMMAND
 105    104 root      R    27912   6%  61% ffmpeg -i track2.wav
 [...]
```

The summary line shows the percentage of time spent running in various states, as shown in this table:

procs	Busybox	
us	usr	User space programs with default nice value
sy	sys	Kernel code
ni	nic	User space programs with non-default nice value
id	idle	Idle
wa	io	I/O wait
hi	irq	Hardware interrupts
si	sirq	Software interrupts
st	--	Steal time: only relevant in virtualized environments

In the preceding example, almost all of the time (58%) is spent in user mode, with a small amount (4%) in system mode, so this is a system that is CPU-bound in user space. The first line after the summary shows that just one application is responsible: `ffmpeg`. Any efforts towards reducing CPU usage should be directed there.

Here is another example:

```
Mem: 13128K used, 490088K free, 40K shrd, 0K buff, 2788K cached
CPU:  0% usr  99% sys   0% nic   0% idle   0% io    0% irq    0% sirq
Load average: 0.41 0.11 0.04 2/46 97
 PID  PPID USER      STAT  VSZ %VSZ %CPU COMMAND
  92     82 root      R    2152   0% 100% cat /dev/urandom
[...]
```

This system is spending almost all of the time in kernel space, as a result of `cat` reading from `/dev/urandom`. In this artificial, case, profiling `cat` by itself would not help, but profiling the kernel functions that `cat` calls might be.

The default view of `top` shows only processes, so the CPU usage is the total of all the threads in the process. Press `H` to see information for each thread. Likewise, it aggregates the time across all CPUs. If you are using `procs top`, you can see a summary per CPU by pressing the `1` key.

Imagine that there is a single user space process taking up most of the time and look at how to profile that.

Poor man's profiler

You can profile an application just by using GDB to stop it at arbitrary intervals and see what it is doing. This is the *poor man's profiler*. It is easy to set up and it is one way of gathering profile data.

The procedure is simple and explained here:

1. Attach to the process using `gdbserver` (for a remote debug) or `gdb` (for a native debug). The process stops.
2. Observe the function it stopped in. You can use the `backtrace` GDB command to see the call stack.
3. Type `continue` so that the program resumes.
4. After a while, type `Ctrl + C` to stop it again and go back to step 2.

If you repeat steps 2 to 4 several times, you will quickly get an idea of whether it is looping or making progress and, if you repeat them often enough, you will get an idea of where the hotspots in the code are.

There is a whole web page dedicated to the idea at <http://poormansprofiler.org>, together with scripts which make it a little easier. I have used this technique many times over the years with various operating systems and debuggers.

This is an example of statistical profiling, in which you sample the program state at intervals. After a number of samples, you begin to learn the statistical likelihood of the functions being executed. It is surprising how few you really need. Other statistical profilers are `perf record`, `OProfile`, and `gprof`.

Sampling using a debugger is intrusive because the program is stopped for a significant period while you collect the sample. Other tools can do that with much lower overhead.

I will now consider how to use `perf` to do statistical profiling.

Introducing `perf`

`perf` is an abbreviation of the **Linux performance event counter subsystem**, `perf_events`, and also the name of the command-line tool for interacting with `perf_events`. Both have been part of the kernel since Linux 2.6.31. There is plenty of useful information in the Linux source tree in `tools/perf/Documentation`, and also at <https://perf.wiki.kernel.org>.

The initial impetus for developing `perf` was to provide a unified way to access the registers of the **performance measurement unit (PMU)**, which is part of most modern processor cores. Once the API was defined and integrated into Linux, it became logical to extend it to cover other types of performance counters.

At its heart, `perf` is a collection of event counters with rules about when they actively collect data. By setting the rules, you can capture data from the whole system, or just the kernel, or just one process and its children, and do it across all CPUs or just one CPU. It is very flexible. With this one tool you can start by looking at the whole system, then zero in on a device driver that seems to be causing problems, or an application that is running slowly, or a library function that seems to be taking longer to execute than you thought.

The code for the `perf` command-line tool is part of the kernel, in the `tools/perf` directory. The tool and the kernel subsystem are developed hand-in-hand, meaning that they must be from the same version of the kernel. `perf` can do a lot. In this chapter, I will examine it only as a profiler. For a description of its other capabilities, read the `perf` man pages and refer to the documentation mentioned in the previous paragraph.

Configuring the kernel for perf

You need a kernel that is configured for `perf_events` and you need the `perf` command cross compiled to run on the target. The relevant kernel configuration is `CONFIG_PERF_EVENTS` present in the menu **General setup | Kernel Performance Events And Counters**.

If you want to profile using tracepoints – more on this subject later – also enable the options described in the section about Ftrace. While you are there, it is worthwhile enabling `CONFIG_DEBUG_INFO` as well.

The `perf` command has many dependencies which makes cross compiling it quite messy. However, both the Yocto Project and Buildroot have target packages for it.

You will also need debug symbols on the target for the binaries that you are interested in profiling, otherwise `perf` will not be able to resolve addresses to meaningful symbols. Ideally, you want debug symbols for the whole system including the kernel. For the latter, remember that the debug symbols for the kernel are in the `vmlinux` file.

Building perf with the Yocto Project

If you are using the standard linux-yocto kernel, `perf_events` is enabled already, so there is nothing more to do.

To build the `perf` tool, you can add it explicitly to the target image dependencies, or you can add the `tools-profile` feature which also brings in `gprof`. As I mentioned previously, you will probably want debug symbols on the target image, and also the kernel `vmlinux` image. In total, this is what you will need in `conf/local.conf`:

```
EXTRA_IMAGE_FEATURES = "debug-tweaks dbg-pkgs tools-profile"
IMAGE_INSTALL_append = " kernel-vmlinux"
```

Building perf with Buildroot

Many Buildroot kernel configurations do not include `perf_events`, so you should begin by checking that your kernel includes the options mentioned in the preceding section.

To cross compile `perf`, run the Buildroot `menuconfig` and select the following:

- `BR2_LINUX_KERNEL_TOOL_PERF` in **Kernel | Linux Kernel Tools**. To build packages with debug symbols and install them unstripped on the target, select these two settings.
- `BR2_ENABLE_DEBUG` in the menu **Build options | build packages with debugging symbols** menu.
- `BR2_STRIP = none` in the menu **Build options | strip command for binaries on target**.

Then, run `make clean`, followed by `make`.

When you have built everything, you will have to copy `vmlinux` into the target image manually.

Profiling with perf

You can use `perf` to sample the state of a program using one of the event counters and accumulate samples over a period of time to create a profile. This is another example of statistical profiling. The default event counter is called `cycles`, which is a generic hardware counter that is mapped to a PMU register representing a count of cycles at the core clock frequency.

Creating a profile using `perf` is a two stage process: the `perf record` command captures samples and writes them to a file named `perf.data` (by default) and then `perf report` analyzes the results. Both commands are run on the target. The samples being collected are filtered for the process and its children, for a command you specify. Here is an example profiling a shell script that searches for the string `linux`:

```
# perf record sh -c "find /usr/share | xargs grep linux > /dev/null"
[ perf record: Woken up 2 times to write data ]
[ perf record: Captured and wrote 0.368 MB perf.data (~16057 samples) ]
# ls -l perf.data
-rw----- 1 root      root      387360 Aug 25 2015 perf.data
```

Now you can show the results from `perf.data` using the command `perf report`. There are three user interfaces which you can select on the command line:

- `--stdio`: This is a pure text interface with no user interaction. You will have to launch `perf report` and annotate for each view of the trace.
- `--tui`: This is a simple text-based menu interface with traversal between screens.
- `--gtk`: This is a graphical interface that otherwise acts in the same way as `--tui`.

The default is TUI, as shown in this example:

Samples: 9K of event 'cycles', Event count (approx.): 2006177260

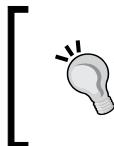
Time (%)	Function	Symbol
11.29%	grep	libc-2.20.so
8.80%	grep	busybox.nosuid
5.55%	grep	libc-2.20.so
5.40%	grep	libc-2.20.so
3.74%	grep	libc-2.20.so
2.59%	grep	libc-2.20.so
2.51%	grep	libc-2.20.so
1.64%	grep	busybox.nosuid
1.57%	grep	libc-2.20.so
1.33%	grep	libc-2.20.so
1.33%	grep	libc-2.20.so
1.26%	grep	[kernel.kallsyms]
1.20%	grep	libc-2.20.so
1.10%	grep	libc-2.20.so
0.95%	grep	libc-2.20.so
0.79%	grep	busybox.nosuid
0.75%	grep	[kernel.kallsyms]
0.72%	grep	[kernel.kallsyms]
0.68%	find	[kernel.kallsyms]
0.53%	grep	[kernel.kallsyms]
0.52%	grep	[kernel.kallsyms]
0.47%	grep	[kernel.kallsyms]

Press '?' for help on key bindings

`perf` is able to record the kernel functions executed on behalf of the processes because it collects samples in kernel space.

The list is ordered with the most active functions first. In this example, all but one are captured while `grep` is running. Some are in a library, `libc-2.20`, some in a program, `busybox.nosuid`, and some are in the kernel. We have symbol names for program and library functions because all the binaries have been installed on the target with debug information, and kernel symbols are being read from `/boot/vmlinux`. If you have `vmlinux` in a different location, add `-k <path>` to the `perf report` command. Rather than storing samples in `perf.data`, you can save them to a different file using `perf record -o <file name>` and analyze them using `perf report -i <file name>`.

By default, `perf record` samples at a frequency of 1000Hz using the cycles counter.



A sampling frequency of 1000Hz may be higher than you really need, and may be the cause of an observer effect. Try with lower rates: 100Hz is enough for most cases, in my experience. You can set the sample frequency using the `-F` option.

Call graphs

This is still not really making life easy; the functions at the top of the list are mostly low level memory operations and you can be fairly sure that they have already been optimized. It would be nice to step back and see where these functions are being called from. You can do that by capturing the backtrace from each sample, which you can do with the `-g` option to `perf record`.

Now `perf report` shows a plus sign (+) where the function is part of a call chain. You can expand the trace to see the functions lower down in the chain:

```
Samples: 10K of event 'cycles', Event count (approx.): 2256721655
- 9.95% grep libc-2.20.so      [.] re_search_internal
- re_search_internal
  95.96% 0
  3.50% 0x208
+ 8.19% grep busybox.nosuid   [.] bb_get_chunk_from_file
+ 5.07% grep libc-2.20.so     [.] _int_free
+ 4.76% grep libc-2.20.so     [.] _int_malloc
+ 3.75% grep libc-2.20.so     [.] realloc
+ 2.63% grep libc-2.20.so     [.] malloc
+ 2.04% grep libc-2.20.so     [.] regexec@@GLIBC_2.4
+ 1.43% grep busybox.nosuid   [.] grep_file
+ 1.37% grep libc-2.20.so     [.] memset
+ 1.29% grep libc-2.20.so     [.] malloc_consolidate
+ 1.22% grep libc-2.20.so     [.] _int_realloc
+ 1.15% grep libc-2.20.so     [.] free
+ 1.01% grep [kernel.kallsyms] [k] __copy_to_user_std
+ 0.98% grep libc-2.20.so     [.] strlen
+ 0.89% grep libc-2.20.so     [.] re_string_reconstruct
+ 0.73% grep [kernel.kallsyms] [k] preempt_count_sub
+ 0.68% grep [kernel.kallsyms] [k] finish_task_switch.isra.85
+ 0.62% grep busybox.nosuid   [.] xrealloc
+ 0.57% grep [kernel.kallsyms] [k] __do_softirq
Press '?' for help on key bindings
```

[ Generating call graphs relies on the ability to extract call frames from the stack, just as is necessary for backtraces in GDB. The information needed to unwind stacks is encoded in the debug information of the executables but not all combinations of architecture and toolchains are capable of doing so.]

perf annotate

Now that you know which functions to look at, it would be nice to step inside and see the code and to have hit counts for each instruction. That is what `perf annotate` does, by calling down to a copy of `objdump` installed on the target. You just need to use `perf annotate` in place of `perf report`.

`perf annotate` requires symbol tables for the executables and `vmlinux`. Here is an example of an annotated function:

```
re_search_internal /lib/libc-2.20.so
    cmp    r1,
    beq    c362c <gai_strerror+0xcaf8>
    str    r3, [fp, #-40]           ; 0x28
    b     c3684 <gai_strerror+0xcb50>
    ldr    ip, [fp, #-256]         ; 0x100
    ldr    r0, [fp, #-268]         ; 0x10c
    add    r3,
    cmp    r0,
    strle r3, [fp, #-40]           ; 0x28
    ble    c3684 <gai_strerror+0xcb50>
    ldrb   r1, [r2, #1]!
    ldrb   r1, [ip, r1]
    cmp    r1,
    beq    c3660 <gai_strerror+0xcb2c>
    str    r3, [fp, #-40]           ; 0x28
    ldr    r3, [fp, #-40]           ; 0x28
    ldr    r2, [fp, #-268]         ; 0x10c
    cmp    r2,
    bne    c3804 <gai_strerror+0xccd0>
    mov    r3,
    ldr    r2, [fp, #-280]         ; 0x118
    cmp    r3,
Press 'h' for help on key bindings
```

If you want to see the source code interleaved with the assembler, you can copy the relevant parts to the target device. If you are using the Yocto Project and build with the extra image feature `dbg-pkgs`, or have installed the individual `-dbg` package, then the source will have been installed for you in `/usr/src/debug`. Otherwise, you can examine the debug information to see the location of the source code:

```
$ arm-buildroot-linux-gnueabi-objdump --dwarf lib/libc-2.19.so |
grep DW_AT_comp_dir
<3f> DW_AT_comp_dir : /home/chris/buildroot/output/build/host-
gcc-initial-4.8.3/build/arm-buildroot-linux-gnueabi/libgcc
```

The path on the target should be exactly the same as the path you can see in `DW_AT_comp_dir`.

Here is an example of annotation with source and assembler code:

```
re_search_internal /lib/libc-2.20.so
    ++match_first;
    goto forward_match_found_start_or_reached_end;

    case 6:
        /* Fastmap without translation, match forward. */
        while (BE (match_first < right_lim, 1)
4.15      cmp    r0,
3.91      strle r3, [fp, #-40]           ; 0x28
        ble    c3684 <gai_strerror+0xcb50>
4.72      && !fastmap[(unsigned char) string[match_first]]])
10.26     ldrb   r1, [r2, #1]!
6.68      ldrb   r1, [ip, r1]
        cmp    r1,
        beq    c3660 <gai_strerror+0xcb2c>
0.90      str    r3, [fp, #-40]           ; 0x28
        ++match_first;

        forward_match_found_start_or_reached_end:
        if (BE (match_first == right_lim, 0))
2.12      ldr    r3, [fp, #-40]           ; 0x28
0.08      ldr    r2, [fp, #-268]         ; 0x10c
0.33      cmp    r2,
Press 'h' for help on key bindings
```

Other profilers: OProfile and gprof

These two statistical profilers predate perf. They are both subsets of the functionality of perf, but they are still quite popular. I will mention them only briefly.

OProfile is a kernel profiler that started out in 2002. Originally, it had its own kernel sampling code, but recent versions use the `perf_events` infrastructure for that purpose. There is more information about it at <http://oprofile.sourceforge.net>. OProfile consists of a kernel-space component and a user space daemon and analysis commands.

OProfile needs these two kernel options to be enabled:

- `CONFIG_PROFILING` in **General setup | Profiling support**
- `CONFIG_OPROFILE` in **General setup | OProfile system profiling**

If you are using the Yocto Project, the user-space components are installed as part of the `tools-profile` image feature. If you are using Buildroot, the package is enabled by `BR2_PACKAGE_OPROFILE`.

You can collect samples by using this command:

```
# operv <program>
```

Wait for your application to finish, or press *Ctrl + C*, to stop profiling. The profile data is stored in `<cur-dir>/oprofile_data/samples/current`.

Use `opreport` to generate a profile summary. There are various options which are documented in the OProfile manual.

`gprof` is part of the GNU toolchain and was one of the earliest open source code profiling tools. It combines compile-time instrumentation and sampling techniques, using a 100 Hz sample rate. It has the advantage that it does not require kernel support.

To prepare a program for profiling with `gprof`, you add `-pg` to the compile and link flags, which injects code that collects information about the call tree into the function preamble. When you run the program, samples are collected and stored in a buffer, which is written to a file named `gmon.out`, when the program terminates.

You use the `gprof` command to read the samples from `gmon.out` and the debug information from a copy of the program.

As an example, if you wanted to profile the BusyBox grep applet, you would rebuild BusyBox with the -pg option, run the command, and view the results:

```
# busybox grep "linux" *
# ls -l gmon.out
-rw-r--r-- 1 root root 473 Nov 24 14:07 gmon.out
```

Then, you would analyze the captured samples on either the target or the host, using the following:

```
# gprof busybox
Flat profile:

Each sample counts as 0.01 seconds.
no time accumulated

%      cumulative      self              self      total
time      seconds      seconds      calls  Ts/call  Ts/call  name
0.00      0.00      0.00          688      0.00      0.00  xrealloc
0.00      0.00      0.00          345      0.00      0.00  bb_get_chunk_from_file
0.00      0.00      0.00          345      0.00      0.00  xmalloc_fgetline
0.00      0.00      0.00            6      0.00      0.00  fclose_if_not_stdin
0.00      0.00      0.00            6      0.00      0.00  fopen_for_read
0.00      0.00      0.00            6      0.00      0.00  grep_file
[...]
Call graph

granularity: each sample hit covers 2 byte(s) no time propagated

index % time      self  children      called      name
        0.00      0.00      688/688  bb_get_chunk_from_file [2]
[1]     0.0      0.00      0.00      688      xrealloc [1]
-----
        0.00      0.00      345/345  xmalloc_fgetline [3]
```

Profiling and Tracing

```
[2]      0.0    0.00   0.00     345    bb_get_chunk_from_file [2]
          0.00   0.00     688/688  xrealloc [1]
-----
          0.00   0.00     345/345  grep_file [6]
[3]      0.0    0.00   0.00     345    xmalloc_fgetline [3]
          0.00   0.00     345/345  bb_get_chunk_from_file
[2]
-----
          0.00   0.00     6/6    grep_main [12]
[4]      0.0    0.00   0.00      6    fclose_if_not_stdin
[4]
[...]
```

Note that the execution times are all shown as zero, because most of the time was spent in system calls, which are not traced by `gprof`.



`gprof` does not capture samples from threads other than the main thread of a multi-threaded process, and it does not sample kernel space, all of which limits its usefulness.



Tracing events

The tools we have seen so far all use statistical sampling. You often want to know more about the ordering of events so that you can see them and relate them to each other. Function tracing involves instrumenting the code with trace points which capture information about the event, and may include some or all of the following:

- Timestamp
- Context, such as the current PID
- Function parameters and return value
- Callstack

It is more intrusive than statistical profiling and it can generate a large amount of data. The latter can be mitigated by applying filters when the sample is captured, and later on when viewing the trace.

I will cover two trace tools here: the kernel function tracers, `Ftrace` and `LTTng`.

Introducing Ftrace

The kernel function tracer, Ftrace, evolved from work done by Steven Rostedt, and many others, as they were tracking down the causes of high latency. Ftrace appeared in Linux 2.6.27 and has been actively developed since then. There are a number of documents describing kernel tracing in the kernel source in Documentation/trace.

Ftrace consists of a number of tracers that can log various types of activity in the kernel. Here, I am going to talk about the function and function_graph tracers, and about the event tracepoints. In *Chapter 14, Real-time Programming*, I will revisit Ftrace and use it to show real-time latencies.

The function tracer instruments each kernel function so that calls can be recorded and timestamped. As a matter of interest, it compiles the kernel with the -pg switch to inject the instrumentation, but there the resemblance to gprof ends. The function_graph tracer goes further and records both the entry and exit of functions so that it can create a call graph. The event tracepoints feature also records parameters associated with the call.

Ftrace has a very embedded-friendly user interface that is entirely implemented through virtual files in the debugfs filesystem, meaning that you do not have to install any tools on the target to make it work. Nevertheless, there are other user interfaces if you prefer: trace-cmd is a command-line tool which records and views traces and is available in Buildroot (BR2_PACKAGE_TRACE_CMD) and the Yocto Project (trace-cmd). There is a graphical trace viewer named KernelShark which is available as a package for the Yocto Project.

Preparing to use Ftrace

Ftrace and its various options are configured in the kernel configuration menu. You will need the following as a minimum:

- CONFIG_FUNCTION_TRACER in the menu **Kernel hacking | Tracers | Kernel Function Tracer**

For reasons that will become clear later, you would be well advised to turn on these options as well:

- CONFIG_FUNCTION_GRAPH_TRACER in the menu **Kernel hacking | Tracers | Kernel Function Graph Tracer**
- CONFIG_DYNAMIC_FTRACE in the menu **Kernel hacking | Tracers | enable/disable function tracing dynamically**

Since the whole thing is hosted in the kernel, there is no user space configuration to be done.

Using Ftrace

Before you can use Ftrace, you have to mount the `debugfs` filesystem which, by convention, goes in the `/sys/kernel/debug` directory:

```
# mount -t debugfs none /sys/kernel/debug
```

All the controls for Ftrace are in the `/sys/kernel/debug/tracing` directory; there is even a mini HOWTO in the `README` file.

This is the list of tracers available in the kernel:

```
# cat /sys/kernel/debug/tracing/available_tracers
blk function_graph function nop
```

The active tracer is shown by `current_tracer`, which, initially, will be the null tracer, `nop`.

To capture a trace, select the tracer by writing the name of one of the `available_tracers` to `current_tracer`, then enable tracing for a short while, as shown here:

```
# echo function > /sys/kernel/debug/tracing/current_tracer
# echo 1 > /sys/kernel/debug/tracing/tracing_on
# sleep 1
# echo 0 > /sys/kernel/debug/tracing/tracing_on
```

In that one second, the trace buffer will have been filled with the details of every function called by the kernel. The format of the trace buffer is plain text, as described in `Documentation/trace/ftrace.txt`. You can read the trace buffer from the `trace` file:

```
# cat /sys/kernel/debug/tracing/trace
# tracer: function
#
# entries-in-buffer/entries-written: 40051/40051    #P:1
#
#                                _----=> irqs-off
#                                / _----=> need_resched
#                                | / _---=> hardirq/softirq
#                                || / _---=> preempt_depth
```

```

#                                ||| /      delay
#      TASK-PID    CPU#  |||   TIMESTAMP   FUNCTION
#      | |       | | |   |       |
sh-361  [000] ...1  992.990646: mutex_unlock <-
rb_simple_write
sh-361  [000] ...1  992.990658: __fsnotify_parent
<-vfs_write
sh-361  [000] ...1  992.990661: fsnotify <-
vfs_write
sh-361  [000] ...1  992.990663: __srcu_read_lock
<-fsnotify
sh-361  [000] ...1  992.990666: preempt_count_add
<-__srcu_read_lock
sh-361  [000] ...2  992.990668: preempt_count_sub
<-__srcu_read_lock
sh-361  [000] ...1  992.990670: __srcu_read_unlock
<-fsnotify
sh-361  [000] ...1  992.990672: __sb_end_write <-
vfs_write
sh-361  [000] ...1  992.990674: preempt_count_add
<-__sb_end_write
[...]

```

You can capture a large number of data points in just one second.

As with profilers, it is difficult to make sense of a flat function list like this. If you select the `function_graph` tracer, Ftrace captures call graphs like this:

```

# tracer: function_graph
#
# CPU DURATION           FUNCTION CALLS
#|   |   |           |   |   |   |
0) + 63.167 us  |           } /* cpdma_ctlr_int_ctrl */
0) + 73.417 us  |           } /* cpsw_intr_disable */
0)               |           disable_irq_nosync() {
0)               |           __disable_irq_nosync() {
0)               |           __irq_get_desc_lock() {
0)   0.541 us    |           irq_to_desc();
0)   0.500 us    |           preempt_count_add();
0) + 16.000 us  |           }

```

```
0)          |           __disable_irq() {  
0)  0.500 us  |               irq_disable();  
0)  8.208 us  |           }  
0)          |           __irq_put_desc_unlock() {  
0)  0.459 us  |               preempt_count_sub();  
0)  8.000 us  |           }  
0) + 55.625 us |       }  
0) + 63.375 us |   }
```

Now you can see the nesting of the function calls, delimited by parentheses, { and }. At the terminating brace, there is a measurement of the time taken in the function, annotated with a plus sign, +, if it takes more than 10 μ s, and an exclamation mark, !, if it takes more than 100 μ s.

You are often only interested in the kernel activity caused by a single process or thread, in which case you can restrict the trace to the one thread by writing the thread ID to `set_ftrace_pid`.

Dynamic Ftrace and trace filters

Enabling `CONFIG_DYNAMIC_FTRACE` allows Ftrace to modify the function trace sites at runtime, which has a couple of benefits. Firstly, it triggers additional build-time processing of the trace function probes which allows the Ftrace subsystem to locate them at boot time and overwrite them with NOP instructions, thus reducing the overhead of the function trace code to almost nothing. You can then enable Ftrace in production or near production kernels with no impact on performance.

The second advantage is that you can selectively enable function trace sites rather than trace everything. The list of functions is put into `available_filter_functions`; there are several tens of thousands of them. You can selectively enable function traces as you need them by copying the name from `available_filter_functions` to `set_ftrace_filter`, and then stop tracing that function by writing the name to `set_ftrace_notrace`. You can also use wildcards and append names to the list. For example, suppose you are interested in `tcp` handling:

```
# cd /sys/kernel/debug/tracing  
# echo "tcp*" > set_ftrace_filter  
# echo function > current_tracer  
# echo 1 > tracing_on
```

Run some tests and then look at the trace:

```
# cat trace
# tracer: function
#
# entries-in-buffer/entries-written: 590/590    #P:1
#
#           _-----> irqs-off
#           / _----> need-resched
#           | / _---> hardirq/softirq
#           || / _--> preempt-depth
#           ||| /     delay
#
#      TASK-PID  CPU#  ||||  TIMESTAMP  FUNCTION
#      | |       |  |||  |       |
dropbear-375  [000] ...1 48545.022235: tcp_poll <-sock_poll
dropbear-375  [000] ...1 48545.022372: tcp_poll <-sock_poll
dropbear-375  [000] ...1 48545.022393: tcp_sendmsg <-
    inet_sendmsg
dropbear-375  [000] ...1 48545.022398: tcp_send_mss <-
    tcp_sendmsg
dropbear-375  [000] ...1 48545.022400: tcp_current_mss <-
    tcp_send_mss
[...]
```

`set_ftrace_filter` can also contain commands, for example, to start and stop tracing when certain functions are executed. There isn't space to go into these details here but, if you want to find out more, please read the **Filter commands** section in Documentation/trace/ftrace.txt.

Trace events

The function and `function_graph` tracers described in the preceding section record only the time at which the function was executed. The trace events feature also records parameters associated with the call, making the trace more readable and informative. For example, instead of just recording that the function `kmalloc` had been called, a trace event will record the number of bytes requested and the returned pointer. Trace events are used in perf and LTTng as well as Ftrace, but the development of the trace events subsystem was prompted by the LTTng project.

It takes effort from kernel developers to create trace events since each one is different. They are defined in the source code using the TRACE_EVENT macro: there are over a thousand of them now. You can see the list of events available at runtime in /sys/kernel/debug/tracing/available_events. They are named subsystem:function, for example, kmem:kmalloc. Each event is also represented by a subdirectory in tracing/events/[subsystem]/[function], as demonstrated here:

```
# ls events/kmem/kmalloc
enable filter format id trigger
```

The files are as follows:

- enable: You write a 1 to this file to enable the event.
- filter: This is an expression which must evaluate to true for the event to be traced.
- format: This is the format of the event and parameters.
- id: This is a numeric identifier.
- trigger: This is a command that is executed when the event occurs using the syntax defined in the **Filter commands** section of Documentation/trace/ftrace.txt. I will show you a simple example involving kmalloc and kfree.

Event tracing does not depend on the function tracers, so begin by selecting the nop tracer:

```
# echo nop > current_tracer
```

Next, select the events to trace by enabling each one individually:

```
# echo 1 > events/kmem/kmalloc/enable
# echo 1 > events/kmem/kfree/enable
```

You can also write the event names to set_event, as shown here:

```
# echo "kmem:kmalloc kmem:kfree" > set_event
```

Now, when you read the trace, you can see the functions and their parameters:

```
# tracer: nop
#
# entries-in-buffer/entries-written: 359/359    #P:1
#
#                                     _----=> irqs-off
#                                     / _----=> need-resched
#                                     | / _---=> hardirq/softirq
#                                     || / _---=> preempt-depth
```

```
#                                ||| /      delay
#      TASK-PID    CPU#  ||||   TIMESTAMP   FUNCTION
#      | | | | | | | | | |
cat-382  [000] ...1 2935.586706: kmalloc:
call_site=c0554644 ptr=de515a00 bytes_req=384 bytes_alloc=512
gfp_flags=GFP_ATOMIC|GFP_NOWARN|GFP_NOMEMALLOC
cat-382  [000] ...1 2935.586718: kfree:
call_site=c059c2d8 ptr= (null)
```

Exactly the same trace events are visible in perf as *tracepoint events*.

Using LTTng

The Linux Trace Toolkit project was started by Karim Yaghmour as a means of tracing kernel activity and was one of the first trace tools generally available for the Linux kernel. Later, Mathieu Desnoyers took up the idea and re-implemented it as the next generation trace tool, LTTng. It was then expanded to cover user space traces as well as the kernel. The project website is at <http://lttng.org/> and contains a comprehensive user manual.

LTTng consists of three components:

- A core session manager
- A kernel tracer implemented as a group of kernel modules
- A user space tracer implemented as a library

In addition to those, you will need a trace viewer such as Babeltrace (<http://www.efficios.com/babeltrace>) or the Eclipse Trace Compaas plug-in to display and filter the raw trace data on the host or target.

LTTng requires a kernel configured with `CONFIG_TRACEPOINTS`, which is enabled when you select **Kernel hacking | Tracers | Kernel Function Tracer**.

The description that follows refers to LTTng version 2.5; other versions may be different.

LTTng and the Yocto Project

You need to add these packages to the target dependencies, for example, in `conf/local.conf`:

```
IMAGE_INSTALL_append = " lttng-tools lttng-modules lttng-ust"
```

If you want to run Babeltrace on the target, also append the package `babeltrace`.

LTTng and Buildroot

You need to enable the following:

- BR2_PACKAGE_LTTNG_MODULES in the menu **Target packages | Debugging, profiling and benchmark | lttng-modules**.
- BR2_PACKAGE_LTTNG_TOOLS in the menu **Target packages | Debugging, profiling and benchmark | lttng-tools**.

For user space trace tracing, enable this:

- BR2_PACKAGE_LTTNG_LIBUST in the menu **Target packages | Libraries | Other, enable lttng-libust**.

There is a package called `lttng-babletrace` for the target. Buildroot builds the host `babeltrace` automatically and places in `output/host/usr/bin/babeltrace`.

Using LTTng for kernel tracing

LTTng can use the set of ftrace events described above as potential trace points. Initially, they are disabled.

The control interface for LTTng is the `lttng` command. You can list the kernel probes using the following:

```
# lttng list --kernel
Kernel events:
-----
        writeback_nothread (loglevel: TRACE_EMERG (0)) (type:
tracepoint)
        writeback_queue (loglevel: TRACE_EMERG (0)) (type:
tracepoint)
        writeback_exec (loglevel: TRACE_EMERG (0)) (type:
tracepoint)
[...]
```

Traces are captured in the context of a session which, in this example, is called `test`:

```
# lttng create test
Session test created.
Traces will be written in /home/root/lttng-traces/test-20150824-
140942
# lttng list
Available tracing sessions:
```

```
1) test (/home/root/lttng-traces/test-20150824-140942)
[inactive]
```

Now enable a few events in the current session. You can enable all kernel tracepoints using the `--all` option but remember the warning about generating too much trace data. Let's start with a couple of scheduler-related trace events:

```
# lttng enable-event --kernel sched_switch,sched_process_fork
```

Check that everything is set up:

```
# lttng list test
Tracing session test: [inactive]
  Trace path: /home/root/lttng-traces/test-20150824-140942
  Live timer interval (usec): 0
```

```
==== Domain: Kernel ===
```

```
Channels:
```

```
-----
- channel0: [enabled]
```

```
Attributes:
```

```
  overwrite mode: 0
  subbufers size: 26214
  number of subbufers: 4
  switch timer interval: 0
  read timer interval: 200000
  trace file count: 0
  trace file size (bytes): 0
  output: splice()
```

```
Events:
```

```
  sched_process_fork (loglevel: TRACE_EMERG (0)) (type: tracepoint) [enabled]
  sched_switch (loglevel: TRACE_EMERG (0)) (type: tracepoint) [enabled]
```

Now start tracing:

```
# lttng start
```

Run the test load and then stop tracing:

```
# lttnng stop
```

Traces for the session are written to the session directory, `lttnng-traces/<session>/kernel`.

You can use the Babeltrace viewer to dump the raw trace data in text format, in this case, I ran it on the host computer:

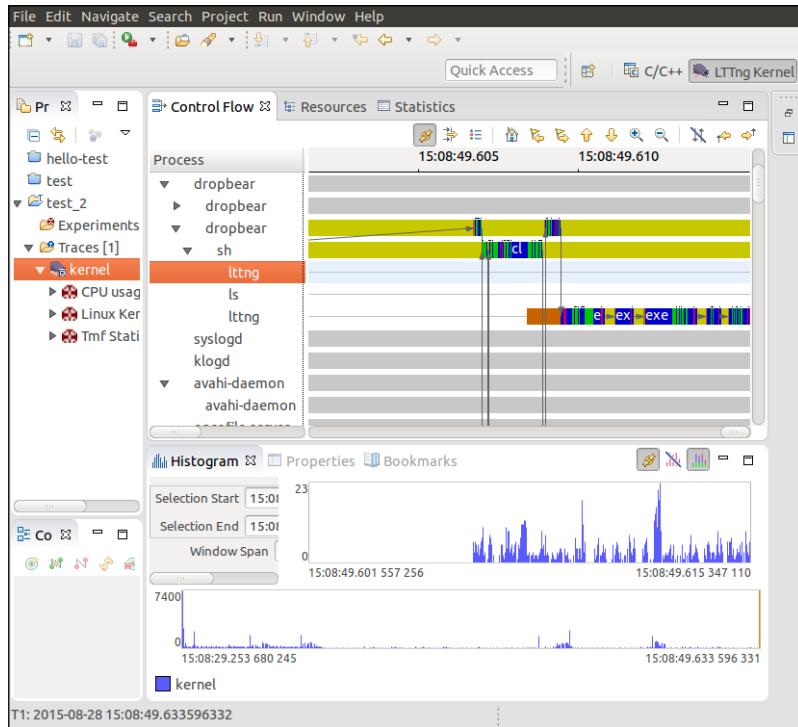
```
$ babeltrace lttnng-traces/test-20150824-140942/kernel
```

The output is too verbose to fit on this page, so I will leave it as an exercise for you, the reader, to capture and display a trace in this way. The text output from eBabeltrace does have the advantage that it is easy to search for strings using grep and similar commands.

A good choice for a graphical trace viewer is the Trace Compass plug-in for Eclipse, which is now part of the Eclipse IDE for C/C++ Developers bundle. Importing the trace data into Eclipse is characteristically fiddly. Briefly, you need to follow these steps:

1. Open the tracing perspective.
2. Create a new project by selecting **File | New | Tracing project**.
3. Enter a project name and click **Finish**.
4. Right-click on the **New Project** option in the **Project Explorer** menu and select **Import**.
5. Expand **Tracing** and then select **Trace Import**.
6. Browse to the directory containing the traces (for example, `test-20150824-140942`), tick the box to indicate which sub-directories you want (it might be the kernel) and click **Finish**.
7. Now, expand the project and, within that, expand **Traces[1]** and, within that, double-click on **kernel**.

8. You should see the trace data shown in the following screenshot:



In the preceding screenshot, I have zoomed in on the control flow view to show state transitions between dropbear and a shell, and also some activity of the `lttng` daemon.

Using Valgrind for application profiling

I introduced Valgrind in *Chapter 11, Managing Memory*, as a tool for identifying memory problems using the memcheck tool. Valgrind has other useful tools for application profiling. The two I am going to look at here are **Callgrind** and **Helgrind**. Since Valgrind works by running the code in a sandbox, it is able to check the code as it runs and report certain behaviors, which native tracers and profilers cannot do.

Callgrind

Callgrind is a call-graph generating profiler that also collects information about processor cache hit rate and branch prediction. Callgrind is only useful if your bottleneck is CPU-bound. It's not useful if heavy I/O or multiple processes are involved.

Valgrind does not require kernel configuration but it does need debug symbols. It is available as a target package in both the Yocto Project and Buildroot (BR2_PACKAGE_VALGRIND).

You run Callgrind in Valgrind on the target, like so:

```
# valgrind --tool=callgrind <program>
```

This produces a file called `callgrind.out.<PID>` which you can copy to the host and analyze with `callgrind_annotate`.

The default is to capture data for all the threads together in a single file. If you add option `--separate-threads=yes` when capturing, there will be profiles for each of the threads in files named `callgrind.out.<PID>-<thread id>`, for example, `callgrind.out.122-01`, `callgrind.out.122-02`, and so on.

Callgrind can simulate the processor L1/L2 cache and report on cache misses. Capture the trace with the `--simulate-cache=yes` option. L2 misses are much more expensive than L1 misses, so pay attention to code with high D2mr or D2mw counts.

Helgrind

This is a thread error detector for detecting synchronization errors in C, C++, and Fortran programs that include POSIX threads.

Helgrind can detect three classes of error. Firstly, it can detect the incorrect use of the API. For example, it can unlock a mutex that is already unlocked, unlock a mutex that was locked by a different thread, not checking the return value of certain Pthread functions. Secondly, it monitors the order in which threads acquire locks and thus detects potential deadlocks which could arise from the formation of cycles of locks. Finally, it detects data races which can happen when two threads access a shared memory location without using suitable locks or other synchronization to ensure single-threaded access.

Using Helgrind is simple, you just need this command:

```
# valgrind --tool=helgrind <program>
```

It prints problems and potential problems as it finds them. You can direct these messages to a file by adding `--log-file=<filename>`.

Using strace to show system calls

I started the chapter with the simple and ubiquitous tool, `top`, and I will finish with another: `strace`. It is a very simple tracer that captures system calls made by a program and, optionally, its children. You can use it to do the following:

- Learn which system calls a program makes.
- Find those system calls that fail together with the error code. I find this useful if a program fails to start but doesn't print an error message or if the message is too general. `strace` shows the failing syscall.
- Find which files a program opens.
- Find out what syscalls a running program is making, for example to see if it is stuck in a loop.

There are many more examples online, just search for `strace` tips and tricks. Everybody has their own favorite story, for example, <http://chadfowler.com/blog/2014/01/26/the-magic-of-strace>

`strace` uses the `ptrace(2)` function to hook calls from user space to the kernel. If you want to know more about how `ptrace` works, the man page is detailed and surprisingly legible.

The simplest way to get a trace is to run the command with `strace` as shown here (the listing has been edited to make it clearer):

```
# strace ./helloworld
execve("./helloworld", ["../helloworld"], /* 14 vars */) = 0
brk(0)                               = 0x11000
uname({sys="Linux", node="beaglebone", ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb6f40000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=8100, ...}) = 0
mmap2(NULL, 8100, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb6f3e000
close(3)                             = 0
open("/lib/tls/v7l/neon/vfp/libc.so.6", O_RDONLY|O_CLOEXEC) = -1
ENOENT (No such file or directory)
[...]
open("/lib/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\1\1\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0$`\1\0004\0\0\0"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1291884, ...}) = 0
```

Profiling and Tracing

```
mmap2(NULL, 1328520, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE,
3, 0) = 0xb6df9000
mprotect(0xb6f30000, 32768, PROT_NONE)   = 0
mmap2(0xb6f38000, 12288, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x137000) = 0xb6f38000
mmap2(0xb6f3b000, 9608, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xb6f3b000
close(3)
[...]
write(1, "Hello, world!\n", 14Hello, world!
)           = 14
exit_group(0)                      = ?
+++ exited with 0 +++
```

Most of the trace shows how the runtime environment is created. In particular you can see how the library loader hunts for `libc.so.6`, eventually finding it in `/lib`. Finally, it gets to running the `main()` function of the program, which prints its message and exits.

If you want `strace` to follow any child processes or threads created by the original process, add the `-f` option.



If you are using `strace` to trace a program that creates threads, you almost certainly want the `-f` option. Better still, use `-ff` and `-o <file name>` so that the output for each child process or thread is written to a separate file named `<filename>. <PID | TID>`.

A common use of `strace` is to discover which files a program tries to open at start up. You can restrict the system calls that are traced through the `-e` option, and you can write the trace to a file instead of `stdout` by using the `-o` option:

```
# strace -e open -o ssh-strace.txt ssh localhost
```

This shows the libraries and configuration files `ssh` opens when it is setting up a connection.

You can even use `strace` as a basic profile tool: if you use the `-c` option, it accumulates the time spent in system calls and prints out a summary like this:

```
# strace -c grep linux /usr/lib/* > /dev/null
% time      seconds  usecs/call    calls    errors syscall
----- -----
78.68      0.012825        1     11098       18      read
```

11.03	0.001798	1	3551		write
10.02	0.001634	8	216	15	open
0.26	0.000043	0	202		fstat64
0.00	0.000000	0	201		close
0.00	0.000000	0	1		execve
0.00	0.000000	0	1	1	access
0.00	0.000000	0	3		brk
0.00	0.000000	0	199		munmap
0.00	0.000000	0	1		uname
0.00	0.000000	0	5		mprotect
0.00	0.000000	0	207		mmap2
0.00	0.000000	0	15	15	stat64
0.00	0.000000	0	1		getuid32
0.00	0.000000	0	1		set_tls
<hr/>					
100.00	0.016300		15702	49	total

Summary

Nobody can complain that Linux lacks options to profile and trace. This chapter has given you an overview of some of the most common ones.

When faced with a system that is not performing as well as you would like, start with `top` and try to identify the problem. If it proves to be a single application, then you can use `perf record/report` to profile it, bearing in mind that you will have to configure the kernel to enable `perf` and you will need debug symbols for the binaries and kernel. OProfile is an alternative to `perf record` and can tell you similar things. `gprof` is, frankly, outdated but it does have the advantage of not requiring kernel support. If the problem is not so well localized, use `perf` (or OProfile) to get a system-wide view.

Ftrace comes into its own when you have specific questions about the behavior of the kernel. The `function` and `function_graph` tracers give a detailed view of the relationship and sequence of function calls. The event tracers allow you to extract more information about functions including the parameters and return values. LTTng performs a similar role, making use of the event trace mechanism, and adds high speed ring buffers to extract large quantities of data from the kernel. Valgrind has the particular advantage that it runs code in a sandbox and can report on errors that are hard to track down in other ways.

Using the Callgrind tool, it can generate call graphs and report on processor cache usage and, with Helgrind, it can report on thread-related problems. Finally, don't forget `strace`. It is a good standby for finding out what system calls a program is making, from tracking file open calls to find file path names to checking for system wake ups and incoming signals.

All the while, be aware of, and try to avoid, the observer effect: make sure that the measurements you are making are valid for a production system. In the next chapter, I will continue the theme as I delve into the latency tracers that help us quantify the real-time performance of a target system.

14

Real-time Programming

Much of the interaction between a computer system and the real world happens in real-time and so this is an important topic for developers of embedded systems. I have touched on real-time programming in several places so far: in *Chapter 10, Learning About Processes and Threads*, I looked at scheduling policies and priority inversion, and in *Chapter 11, Managing Memory*, I described the problems with page faults and the need for memory locking. Now, it is time to bring these topics together and look at real-time programming in some depth.

In this chapter, I will begin with a discussion about the characteristics of real-time systems and then consider the implications for system design, both at the application and kernel levels. I will describe the real-time kernel patch, `PREEMPT_RT`, and show how to get it and apply it to a mainline kernel. The last sections will describe how to characterize system latencies using two tools: `cyclictest` and `Ftrace`.

There are other ways to achieve real-time behavior on an embedded Linux device, for instance, using a dedicated micro-controller or a separate real-time kernel alongside the Linux kernel in the way that Xenomai and RTAI do. I am not going to discuss these here because the focus of this book is on using Linux as the core for embedded systems.

What is real-time?

The nature of real-time programming is one of the subjects that software engineers love to discuss at length, often giving a range of contradictory definitions. I will begin by setting out what I think is important about real-time.

A task is a real-time task if it has to complete before a certain point in time, known as the deadline. The distinction between real-time and non real-time tasks is shown by considering what happens when you play an audio stream on your computer while compiling the Linux kernel.

The first is a real-time task because there is a constant stream of data arriving at the audio driver and blocks of audio samples have to be written to the audio interface at the playback rate. Meanwhile, the compilation is not real-time because there is no deadline. You simply want it to complete as soon as possible; whether it takes 10 seconds or 10 minutes does not affect the quality of the kernel.

The other important thing to consider is the consequence of missing the deadline, which can range from mild annoyance through to system failure and death.

Here are some examples:

- **Playing an audio stream:** There is a deadline in the order of tens of milliseconds. If the audio buffer under-runs you will hear a click, which is annoying, but you will get over it.
- **Moving and clicking a mouse:** The deadline is also in the order of tens of milliseconds. If it is missed, the mouse moves erratically and button clicks will be lost. If the problem persists, the system will become unusable.
- **Printing a piece of paper:** The deadlines for the paper feed are in the millisecond range, which, if missed, may cause the printer to jam and somebody will have to go and fix it. Occasional jams are acceptable but nobody is going to buy a printer that keeps on jamming.
- **Printing sell-by dates on bottles on a production line:** If one bottle is not printed the whole production line has to be halted, the bottle removed and the line restarted, which is expensive.
- **Baking a cake:** There is a deadline of 30 minutes or so. If you miss it by a few minutes, the cake might be ruined. If you miss it by a large amount, the house will burn down.
- **A power surge detection system:** If the system detects a surge, a circuit breaker has to be triggered within 2 milliseconds. Failing to do so causes damage to the equipment and may injure or kill personnel.

In other words, there are many consequences to missed deadlines. We often talk about these different categories:

- **soft real-time:** The deadline is desirable but is sometimes missed without the system being considered a failure. First two examples are like this.
- **hard real-time:** Here, missing a deadline has a serious effect. We can further subdivide hard real-time into mission-critical systems in which there is a cost to missing the deadline, such as the fourth example, and safety critical-systems in which there is a danger to life and limb, such as the last two examples.
I put in the baking example to show that not all hard real-time systems have deadlines measured in microseconds.

Software written for safety-critical systems has to conform to various standards that seek to ensure that it is capable of performing reliably. It is very difficult for a complex operating system such as Linux to meet those requirements.

When it comes to mission-critical systems, it is possible, and common, for Linux to be used for a wide range of control systems. The requirements of the software depend on the combination of the deadline and the confidence level, which can usually be determined through extensive testing.

Therefore, to say that a system is real-time, you have to measure its response times under the maximum anticipated load, and show that it meets the deadline for an agreed proportion of the time. As a rule of thumb, a well configured Linux system using a mainline kernel is good for soft real-time tasks with deadlines down to tens of milliseconds and a kernel with the PREEMPT_RT patch is good for soft and hard real-time mission-critical systems with deadlines down to several hundreds of microseconds.

The key to creating a real-time system is to reduce the variability in response times so that you have greater confidence that they will not be missed; in other words, you need to make the system more deterministic. Often, this is done at the expense of performance. For example, caches make systems run faster by making the average time to access an item of data shorter, but the maximum time is longer in the case of a cache miss. Caches make a system faster but less deterministic, which is the opposite of what we want.



It is a myth of real-time computing that it is fast. This is not so, the more deterministic a system is, the lower the maximum throughput.

The remainder of this chapter is concerned with identifying the causes of latency and the things you can do to reduce it.

Identifying the sources of non-determinism

Fundamentally, real-time programming is about making sure that the threads controlling the output in real-time are scheduled when needed and so can complete the job before the deadline. Anything that prevents this is a problem. Here are some problem areas:

- **Scheduling:** Real-time threads must be scheduled before others so they must have a real-time policy, `SCHED_FIFO`, or `SCHED_RR`. Additionally they should have priorities assigned in descending order starting with the one with the shortest deadline, according to the theory of Rate Monotonic Analysis that I described in *Chapter 10, Learning About Processes and Threads*.
- **Scheduling latency:** The kernel must be able to reschedule as soon as an event such as an interrupt or timer occurs, and not be subject to unbounded delays. Reducing scheduling latency is a key topic later on in this chapter.
- **Priority inversion:** This is a consequence of priority-based scheduling, which leads to unbounded delays when a high priority thread is blocked on a mutex held by a low priority thread, as I described in *Chapter 10, Learning About Processes and Threads*. User space has priority inheritance and priority ceiling mutexes; in kernel space we have rt-mutexes which implement priority inheritance and which I will talk about in the section on the real-time kernel.
- **Accurate timers:** If you want to manage deadlines in the region of low milliseconds or microseconds, you need timers that match. High resolution timers are crucial and are a configuration option on almost all kernels.
- **Page faults:** A page fault while executing a critical section of code will upset all timing estimates. You can avoid them by locking memory, as I describe later on.
- **Interrupts:** They occur at unpredictable times and can result in unexpected processing overhead if there is a sudden flood of them. There are two ways to avoid this. One is to run interrupts as kernel threads, and the other, on multi-core devices, is to shield one or more CPUs from interrupt handling. I will discuss both possibilities later.
- **Processor caches:** Provide a buffer between the CPU and the main memory and, like all caches, are a source of non-determinism, especially on multi-core devices. Unfortunately, this is beyond the scope of this book but, refer to the references at the end of the chapter.

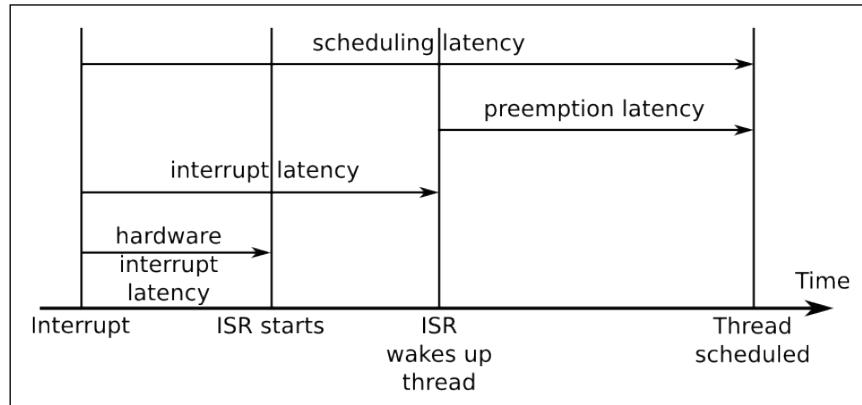
- **Memory bus contention:** When peripherals access memory directly through a DMA channel they use up a slice of memory bus bandwidth, which slows down access from the CPU core (or cores) and so contributes to non-deterministic execution of the program. However, this is a hardware issue and is also beyond the scope of this book.

I will expand on the important problems and see what can be done about them in the next sections.

One item missing from the list is power management. The needs of real-time and power management pull in opposite directions. Power management often leads to high latencies when switching between sleep states, since setting up power regulators and waking up processors all takes time, as does changing the core clock frequency because the clocks take time to settle. But, surely you wouldn't expect a device to respond immediately to an interrupt from suspend state? I know I can't get going in the morning until after at least one cup of coffee.

Understanding scheduling latency

Real-time threads need to be scheduled as soon as they have something to do. However, even if there are no other threads of the same or higher priority, there is always a delay from the point at which the wake up event occurs – an interrupt or system timer – to the time that the thread starts to run. This is called scheduling latency. It can be broken down into several components, as shown in the following diagram:



Firstly, there is the hardware interrupt latency from the point at which an interrupt is asserted until the **ISR (interrupt service routine)** begins to run. A small part of this is the delay in the interrupt hardware itself but the biggest problem is interrupts disabled in the software. Minimizing this *IRQ off time* is important.

The next is interrupt latency, which is the length of time until the ISR has serviced the interrupt and woken up any threads waiting on this event. It is mostly dependent on the way the ISR was written. Normally it should take only a short time, measured in micro-seconds.

The final delay is the preemption latency, which is the time from the point that the kernel is notified that a thread is ready to run to that at which the scheduler actually runs the thread. It is determined by whether the kernel can be preempted or not. If it is running code in a critical section then the reschedule will have to wait. The length of the delay is dependent on the configuration of kernel preemption.

Kernel preemption

The preemption latency occurs because it is not always safe or desirable to preempt the current thread of execution and call the scheduler. Mainline Linux has three settings for preemption, selected via the **Kernel Features | Preemption Model** menu:

- `CONFIG_PREEMPT_NONE`: no preemption
- `CONFIG_PREEMPT_VOLUNTARY`: enables additional checks for requests for preemption
- `CONFIG_PREEMPT`: allows the kernel to be preempted

With preemption set to none, kernel code will continue without rescheduling until it either returns via a `syscall` back to user space, where preemption is always allowed, or it encounters a sleeping wait which stops the current thread. Since it reduces the number of transitions between the kernel and user space and may reduce the total number of context switches, this option results in the highest throughput at the expense of large preemption latencies. It is the default for servers and some desktop kernels where throughput is more important than responsiveness.

The second option enables more explicit preemption points where the scheduler is called if the `need_resched` flag is set, which reduces the worst case preemption latencies at the expense of slightly lower throughput. Some distributions set this option on desktops.

The third option makes the kernel preemptible, meaning that an interrupt can result in an immediate reschedule so long as the kernel is not executing in an atomic context, which I will describe in the following section. This reduces worst case preemption latencies and, therefore, overall scheduling latencies, to something in the order of a few milliseconds on typical embedded hardware. This is often described as a soft real-time option and most embedded kernels are configured in this way. Of course, there is a small reduction in overall throughput but that is usually less important than having more deterministic scheduling for embedded devices.

The real-time Linux kernel (`PREEMPT_RT`)

There is a long-standing effort to reduce latencies still further which goes by the name of the kernel configuration option for these features, `PREEMPT_RT`. The project was started by Ingo Molnar, Thomas Gleixner, and Steven Rostedt and has had contributions from many more developers over the years. The kernel patches are at <https://www.kernel.org/pub/linux/kernel/projects/rt> and there is a wiki, including an FAQ (slightly out of date), at <https://rt.wiki.kernel.org>.

Many parts of the project have been incorporated into mainline Linux over the years, including high resolution timers, kernel mutexes, and threaded interrupt handlers. However, the core patches remain outside of the mainline because they are rather intrusive and (some claim) only benefit a small percentage of the total Linux user base. Maybe, one day, the whole patch set will be merged upstream.

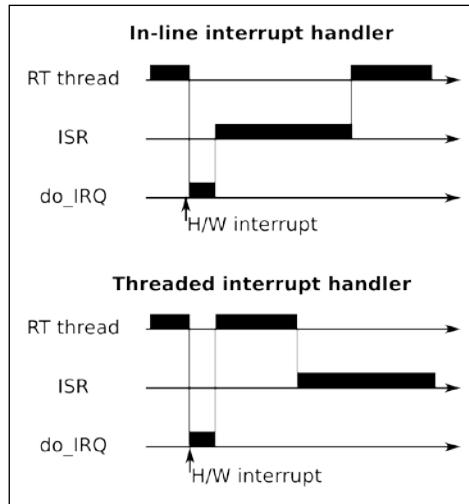
The central plan is to reduce the amount of time the kernel spends running in an atomic context, which is where it is not safe to call the scheduler and switch to a different thread. Typical atomic contexts are when the kernel:

- is running an interrupt or trap handler
- is holding a spin lock or in an RCU critical section. Spin lock and RCU are kernel locking primitives, the details of which are not relevant here
- is between calls to `preempt_disable()` and `preempt_enable()`
- hardware interrupts are disabled

The changes that are part of `PREEMPT_RT` fall into two main areas: one is to reduce the impact of interrupt handlers by turning them into kernel threads and the other is to make locks preemptible so that a thread can sleep while holding one. It is obvious that there is a large overhead in these changes, which makes average case interrupt handling slower but much more deterministic, which is what we are striving for.

Threaded interrupt handlers

Not all interrupts are triggers for the real-time tasks but all interrupts steal cycles from the real-time task. Threaded interrupt handlers allow a priority to be associated with the interrupt and for it to be scheduled at an appropriate time as shown in the following diagram:

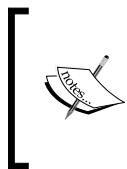


If the interrupt handler code is run as a kernel thread there is no reason why it cannot be preempted by a user space thread of higher priority, and so the interrupt handler does not contribute towards scheduling latency of the user space thread. Threaded interrupt handlers have been a feature of mainline Linux since 2.6.30. You can request that an individual interrupt handler is threaded by registering it with `request_threaded_irq()` in place of the normal `request_irq()`. You can make threaded IRQs the default by configuring the kernel with `CONFIG_IRQ_FORCED_THREADING=y` which makes all handlers into threads unless they have explicitly prevented this by setting the `IRQF_NO_THREAD` flag. When you apply the `PREEMPT_RT` patches, interrupts are, by default, configured as threads in this way. Here is an example of what you might see:

# ps -Leo pid,tid,class,rtprio,stat,comm,wchan grep FF						
PID	TID	CLS	RTPRIO	STAT	COMMAND	WCHAN
3	3	FF	1	S	ksoftirqd/0	smpboot_th
7	7	FF	99	S	posixcpumr/0	posix_cpu_
19	19	FF	50	S	irq/28-edma	irq_thread
20	20	FF	50	S	irq/30-edma_err	irq_thread
42	42	FF	50	S	irq/91-rtc0	irq_thread

43	43	FF	50	S	irq/92-rtc0	irq_thread
44	44	FF	50	S	irq/80-mmc0	irq_thread
45	45	FF	50	S	irq/150-mmc0	irq_thread
47	47	FF	50	S	irq/44-mmc1	irq_thread
52	52	FF	50	S	irq/86-44e0b000	irq_thread
59	59	FF	50	S	irq/52-tilcdc	irq_thread
65	65	FF	50	S	irq/56-4a100000	irq_thread
66	66	FF	50	S	irq/57-4a100000	irq_thread
67	67	FF	50	S	irq/58-4a100000	irq_thread
68	68	FF	50	S	irq/59-4a100000	irq_thread
76	76	FF	50	S	irq/88-OMAP_UAR	irq_thread

In this case, a BeagleBone running `linux-yocto-rt`, only the `gp_timer` interrupt was not threaded. It is normal that the timer interrupt handler be run in-line.



Note that the interrupt threads have all been given the default policy `SCHED_FIFO` and a priority of 50. It doesn't make sense to leave them with the defaults, however; now is your chance to assign priorities according to the importance of the interrupts compared to real-time user space threads.

Here is a suggested order of descending thread priorities:

- The POSIX timers thread, `posixcpumr`, should always have the highest priority.
- Hardware interrupts associated with the highest priority real-time thread.
- The highest priority real-time thread.
- Hardware interrupts for the progressively lower priority real-time threads followed by the thread itself.
- Hardware interrupts for non-real-time interfaces.
- The soft IRQ daemon, `ksoftirqd`, which on RT kernels is responsible for running delayed interrupt routines and, prior to Linux 3.6, was responsible for running the network stack, the block I/O layer, and other things. You may need to experiment with different priority levels to get a balance.

You can change the priorities using the `chrt` command as part of the boot script, using a command like this:

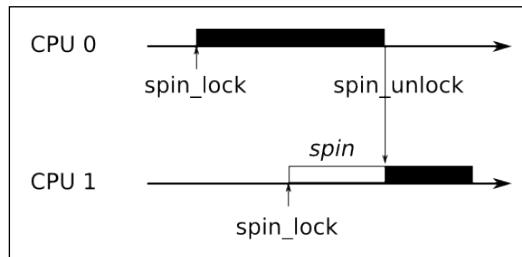
```
# chrt -f -p 90 `pgrep irq/28-edma`
```

The `pgrep` command is part of the `procps` package.

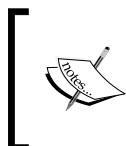
Preemptible kernel locks

Making the majority of kernel locks preemptible is the most intrusive change that PREEMPT_RT makes and this code remains outside of the mainline kernel.

The problem occurs with spinlocks, which are used for much of the kernel locking. A spinlock is a busy-wait mutex which does not require a context switch in the contended case and so is very efficient as long as the lock is held for a short time. Ideally, they should be locked for less than the time it would take to reschedule twice. The following diagram shows threads running on two different CPUs contending the same spinlock. CPU0 gets it first, forcing CPU1 to spin, waiting until it is unlocked:



The thread that holds the spinlock cannot be preempted since doing so may make the new thread enter the same code and deadlock when it tries to lock the same spinlock. Consequently, in mainline Linux, locking a spinlock disables kernel preemption, creating an atomic context. This means that a low priority thread that holds a spinlock can prevent a high priority thread from being scheduled.



The solution adopted by PREEMPT_RT is to replace almost all spinlocks with rt-mutexes. A mutex is slower than a spinlock but it is fully preemptible. Not only that, but rt-mutexes implement priority inheritance and so are not susceptible to priority inversion.



Getting the PREEMPT_RT patches

The RT developers do not create patch sets for every kernel version because of the amount of effort involved. On average, they create patches for every other kernel. The most recent kernels that are supported at the time of writing are as follows:

- 4.1-rt
- 4.0-rt
- 3.18-rt

- 3.14-rt
- 3.12-rt
- 3.10-rt

The patches are available at <https://www.kernel.org/pub/linux/kernel/projects/rt>.

If you are using the Yocto Project, there is an `rt` version of the kernel already. Otherwise, it is possible that the place you got your kernel from already has the `PREEMPT_RT` patch applied. Otherwise, you will have to apply the patch yourself. Firstly, make sure that the `PREEMPT_RT` patch version and your kernel version match exactly, otherwise you will not be able to apply the patches cleanly. Then you apply it in the normal way, as shown here:

```
$ cd linux-4.1.10
$ zcat patch-4.1.10-rt11.patch.gz | patch -p1
```

You will then be able to configure the kernel with `CONFIG_PREEMPT_RT_FULL`.

There is a problem in the last paragraph. The RT patch will only apply if you are using a compatible mainline kernel. You are probably not because that is the nature of embedded Linux kernels and so you will have to spend some time looking at failed patches and fixing them, and then analyzing the board support for your target and adding any real-time support that is missing. These details are, once again, outside the scope of this book. If you are not sure what to do, you should inquire of the developers of the kernel you are using and on kernel developer's forums.

The Yocto Project and PREEMPT_RT

The Yocto Project supplies two standard kernel recipes: `linux-yocto` and `linux-yocto-rt`, the latter having the real-time patches already applied. Assuming that your target is supported by these kernels, then you just need to select `linux-yocto-rt` as your preferred kernel and declare that your machine is compatible, for example, by adding lines similar to these to your `conf/local.conf`:

```
PREFERRED_PROVIDER_virtual/kernel = "linux-yocto-rt"
COMPATIBLE_MACHINE_beaglebone = "beaglebone"
```

High resolution timers

Timer resolution is important if you have precise timing requirements which is typical for real-time applications. The default timer in Linux is a clock that runs at a configurable rate, typically 100 Hz for embedded systems and 250 Hz for servers and desktops. The interval between two timer ticks is known as a **jiffy** and, in the examples given above, is 10 milliseconds on an embedded SoC and four milliseconds on a server.

Linux gained more accurate timers from the real-time kernel project in version 2.6.18 and now they are available on all platforms, providing that there is a high resolution timer source and device driver for it – which is almost always the case. You need to configure the kernel with `CONFIG_HIGH_RES_TIMERS=y`.

With this enabled, all the kernel and user space clocks will be accurate down to the granularity of the underlying hardware. Finding the actual clock granularity is difficult. The obvious answer is the value provided by `clock_getres(2)` but that always claims a resolution of one nanosecond. The `cyclictest` tool that I will describe later has an option to analyze the times reported by the clock to guess the resolution:

```
# cyclictest -R
# /dev/cpu_dma_latency set to 0us
WARN: reported clock resolution: 1 nsec
WARN: measured clock resolution approximately: 708 nsec
You can also look at the kernel log messages for strings like this:
# dmesg | grep clock
OMAP clockevent source: timer2 at 24000000 Hz
sched_clock: 32 bits at 24MHz, resolution 41ns, wraps every
178956969942ns
OMAP clocksource: timer1 at 24000000 Hz
Switched to clocksource timer1
```

The two methods give rather different numbers, for which I have no good explanation but, since both are below one microsecond, I am happy.

Avoiding page faults in a real-time application

A page fault occurs when an application reads or writes memory that is not committed to physical memory. It is impossible (or very hard) to predict when a page fault will happen so they are another source of non-determinism in computers.

Fortunately, there is a function that allows you to commit all memory for a process and lock it down so that it cannot cause a page fault. It is `mlockall(2)`. These are its two flags:

- `MCL_CURRENT`: locks all pages currently mapped
- `MCL_FUTURE`: locks pages that are mapped in later

You usually call `mlockall(2)` during the start up of the application with both flags set to lock all current and future memory mappings.



Note that `MCL_FUTURE` is not magic in that there will still be non-deterministic delay when allocating or freeing heap memory using `malloc()`/`free()` or `mmap()`. Such operations are best done at start up and not in the main control loops.

Memory allocated on the stack is trickier because it is done automatically and if you call a function that makes the stack deeper than before, you will encounter more memory management delays. A simple fix is to grow the stack to a size larger than you think you will ever need at start up. The code would look like this:

```
#define MAX_STACK (512*1024)
static void stack_grow (void)
{
    char dummy[MAX_STACK];
    memset(dummy, 0, MAX_STACK);
    return;
}

int main(int argc, char* argv[])
{
    [...]
    stack_grow ();
    mlockall(MCL_CURRENT | MCL_FUTURE);
    [...]
```

The `stack_grow()` function allocates a large variable on the stack and then zeroes it to force those pages of memory to be committed to this process.

Interrupt shielding

Using threaded interrupt handlers helps mitigate interrupt overhead by running some threads at a higher priority than interrupt handlers that do not impact the real-time tasks. If you are using a multi-core processor, you can take a different approach and shield one or more cores from processing interrupts completely, allowing them to be dedicated to real-time tasks instead. This works either with a normal Linux kernel or a `PREEMPT_RT` kernel.

Achieving this is a question of pinning the real-time threads to one CPU and the interrupt handlers to a different one. You can set the CPU affinity off a thread or process using the command line tool `taskset`, or you can use the `sched_setaffinity(2)` and `pthread_setaffinity_np(3)` functions.

To set the affinity of an interrupt, first note that there is a subdirectory for each interrupt number in `/proc/irq/<IRQ_number>`. The control files for the interrupt are in there, including a CPU mask in `smp_affinity`. Write a bitmask to that file with a bit set for each CPU that is allowed to handle that IRQ.

Measuring scheduling latencies

All the configuration and tuning you may do will be pointless if you cannot show that your device meets the deadlines. You will need your own benchmarks for the final testing but I will describe here two important measurement tools: `cyclictest` and `Ftrace`.

cyclictest

`cyclictest` was originally written by Thomas Gleixner and is now available on most platforms in a package named `rt-tests`. If you are using the Yocto Project, you can create a target image that includes `rt-tests` by building the real-time image recipe like this:

```
$ bitbake core-image-rt
```

If you are using Buildroot, you need to add the package, `BR2_PACKAGE_RT_TESTS` in the menu **Target packages | Debugging, profiling and benchmark | rt-tests**.

`cyclictest` measures scheduling latencies by comparing the actual time taken for a sleep to the requested time. If there was no latency they would be the same and the reported latency would be zero. `cyclictest` assumes a timer resolution of less than one microsecond.

It has a large number of command-line options. To start with, you might try running this command as root on the target:

```
# cyclictest -l 100000 -m -n -p 99
# /dev/cpu_dma_latency set to 0us
policy: fifo: loadavg: 1.14 1.06 1.00 1/49 320

T: 0 ( 320) P:99 I:1000 C: 100000 Min:  9 Act:  13 Avg:  15
Max: 134
```

The options selected are as follows:

- `-l N`: loop N times: the default is unlimited
- `-m`: lock memory with mlockall
- `-n`: use `clock_nanosleep(2)` instead of `nanosleep(2)`
- `-p N`: use the real-time priority N

The result line shows the following, reading from left to right:

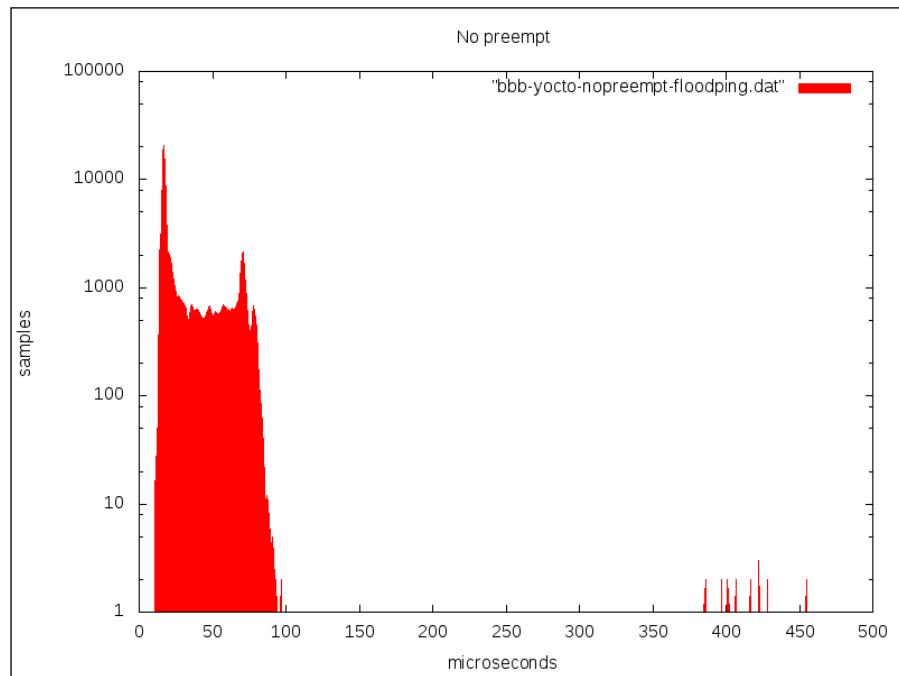
- `T: 0`: this was thread 0, the only thread in this run. You can set the number of threads with parameter `-t`.
- `(320)`: this was PID 320.
- `P:99`: the priority was 99.
- `I:1000`: the interval between loops was 1,000 microseconds. You can set the interval with parameter `-i N`.
- `C:100000`: the final loop count for this thread was 100,000.
- `Min: 9`: the minimum latency was 9 microseconds.
- `Act:13`: the actual latency was 13 microseconds. The actual latency is the most recent latency measurement, which only makes sense if you are watching `cyclictest` run.
- `Avg:15`: the average latency was 15 microseconds.
- `Max:134`: the maximum latency was 134 microseconds.

This was obtained on an idle system running an unmodified `linux-yocto` kernel as a quick demonstration of the tool. To be of real use, you would run tests over a 24 hour period or more while running a load representative of the maximum you expect.

Of the numbers produced by `cyclictest`, the maximum latency is the most interesting, but it would be nice to get an idea of the spread of the values. You can get that by adding `-h <N>` to obtain a histogram of samples that are up to `N` microseconds late. Using this technique, I obtained three traces for the same target board running kernels with no preemption, with standard preemption, and with RT preemption while being loaded with Ethernet traffic from a flood ping. The command line was as shown here:

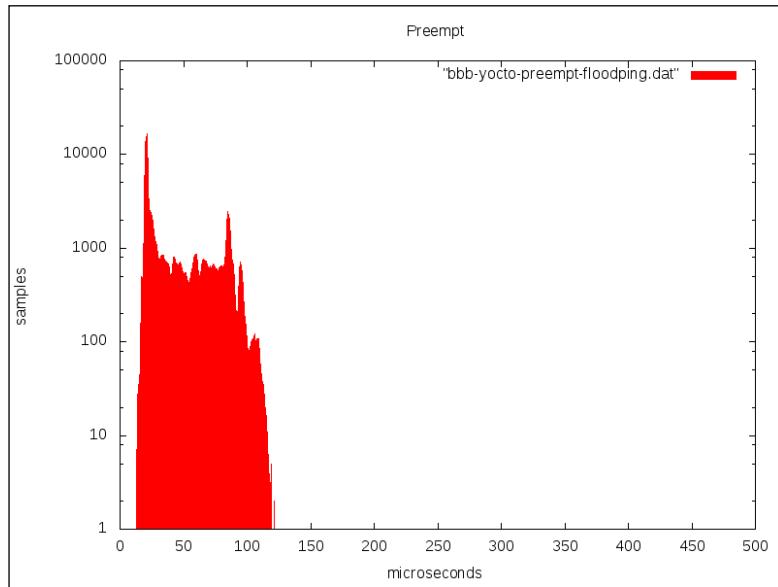
```
# cyclictest -p 99 -m -n -l 100000 -q -h 500 > cyclictest.data
```

The following is the output generated with no preemption:



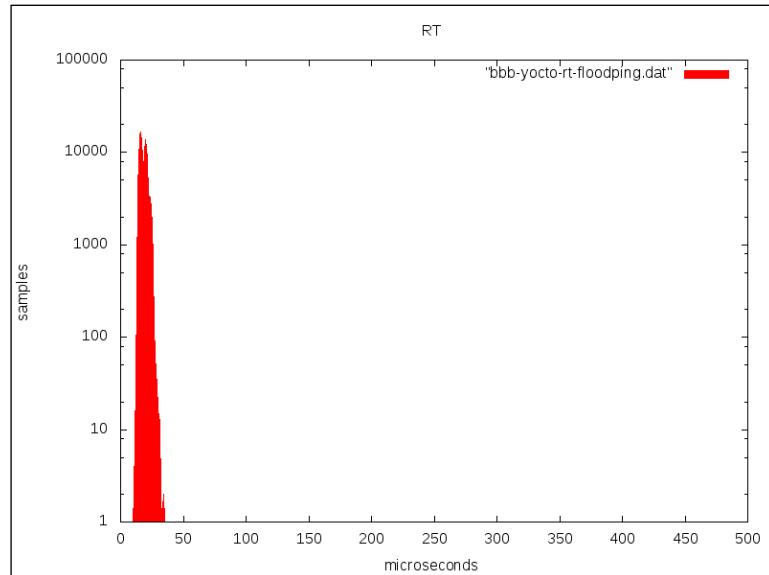
Without preemption, most samples are within 100 microseconds of the deadline, but there are some outliers of up to 500 microseconds, which is pretty much what you would expect.

This is the output generated with standard preemption:



With preemption, the samples are spread out at the lower end but there is nothing beyond 120 microseconds.

Here is the output generated with RT preemption:



The RT kernel is a clear winner because everything is tightly bunched around the 20 microsecond mark and there is nothing later than 35 microseconds.

`cyclictest`, then, is a standard metric for scheduling latencies. However, it cannot help you identify and resolve specific problems with kernel latency. To do that, you need `Ftrace`.

Using Ftrace

The kernel function tracer has tracers to help track down kernel latencies – that is what it was originally written for, after all. These tracers capture the trace for the worst case latency detected during a run, showing the functions that caused the delay. The tracers of interest, together with the kernel configuration parameters, are as follows:

- `irqsoff`: `CONFIG_IRQSOFF_TRACER` traces code that disables interrupts, recording the worst case
- `preemptoff`: `CONFIG_PREEMPT_TRACER` is similar to `irqsoff`, but traces the longest time that kernel preemption is disabled (only available on preemptible kernels)
- `preemptirqsoff`: it combines the previous two traces to record the largest time either `irqs` and/or preemption is disabled
- `wakeup`: traces and records the maximum latency that it takes for the highest priority task to get scheduled after it has been woken up
- `wakeup_rt`: the same as `wake up` but only for real-time threads with the `SCHED_FIFO`, `SCHED_RR`, or `SCHED_DEADLINE` policies
- `wakeup_dl`: the same but only for deadline-scheduled threads with the `SCHED_DEADLINE` policy

Be aware that running `Ftrace` adds a lot of latency, in the order of tens of milliseconds, every time it captures a new maximum which `Ftrace` itself can ignore. However, it skews the results of user-space tracers such as `cyclictest`. In other words, ignore the results of `cyclictest` if you run it while capturing traces.

Selecting the tracer is the same as for the function tracer we looked at in *Chapter 13, Profiling and Tracing*. Here is an example of capturing a trace for the maximum period with preemption disabled for a period of 60 seconds:

```
# echo preemptoff > /sys/kernel/debug/tracing/current_tracer
# echo 0 > /sys/kernel/debug/tracing/tracing_max_latency
# echo 1 > /sys/kernel/debug/tracing/tracing_on
# sleep 60
# echo 0 > /sys/kernel/debug/tracing/tracing_on
```

The resulting trace, heavily edited, looks like this:

```
# cat /sys/kernel/debug/tracing/trace
# tracer: preemptoff
#
# preemptoff latency trace v1.1.5 on 3.14.19-yocto-standard
# -----
# latency: 1160 us, #384/384, CPU#0 | (M:preempt VP:0, KP:0, SP:0 HP:0)
# -----
#     | task: init-1 (uid:0 nice:0 policy:0 rt_prio:0)
# -----
# => started at: ip_finish_output
# => ended at: __local_bh_enable_ip
#
#
#           -----> CPU#
#           / -----> irqs-off
#           | / -----> need-resched
#           || / -----> hardirq/softirq
#           ||| / -----> preempt-depth
#           |||| /     delay
# cmd      pid  ||||| time   |   caller
# \   /  ||||| \   |   /
init-1    0..s.    1us+: ip_finish_output
init-1    0d.s2   27us+: preempt_count_add <-cpdma_chan_submit
init-1    0d.s3   30us+: preempt_count_add <-cpdma_chan_submit
init-1    0d.s4   37us+: preempt_count_sub <-cpdma_chan_submit

[...]

init-1    0d.s2 1152us+: preempt_count_sub <-__local_bh_enable
init-1    0d..2 1155us+: preempt_count_sub <-__local_bh_enable_ip
init-1    0d..1 1158us+: __local_bh_enable_ip
init-1    0d..1 1162us!: trace_preempt_on <-__local_bh_enable_ip
init-1    0d..1 1340us : <stack trace>
```

Here, you can see that the longest period with kernel preemption disabled while running the trace was 1,160 microseconds. This simple fact is available by reading `/sys/kernel/debug/tracing/tracing_max_latency`, but the trace above goes further and gives you the sequence of kernel function calls that lead up to that measurement. The column marked `delay` shows the point on the trail where each function was called, ending with the call to `trace_preempt_on()` at 1162us, at which point kernel preemption is once again enabled. With this information, you can look back through the call chain and (hopefully) work out if this is a problem or not.

The other tracers mentioned work in the same way.

Combining `cyclictest` and `Ftrace`

If `cyclictest` reports unexpectedly long latencies you can use the `breaktrace` option to abort the program and trigger `Ftrace` to obtain more information.

You invoke `breaktrace` using `-b<N>` or `--breaktrace=<N>` where `N` is the number of microseconds of latency that will trigger the trace. You select the `Ftrace` tracer using `-T[tracer name]` or one of the following:

- `-C`: context switch
- `-E`: event
- `-f`: function
- `-w`: wakeup
- `-W`: wakeup-rt

For example, this will trigger the `Ftrace` function tracer when a latency greater than 100 microseconds is measured:

```
# cyclictest -a -t -n -p99 -f -b100
```

Further reading

The following resources have further information about the topics introduced in this chapter:

- *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications* by Buttazzo, Giorgio, Springer, 2011
- *Multicore Application Programming* by Darryl Gove, Addison Wesley, 2011

Summary

The term real-time is meaningless unless you qualify it with a deadline and an acceptable miss rate. When you know that you can determine whether or not Linux is a suitable candidate for the operating system and, if so, begin to tune your system to meet the requirements. Tuning Linux and your application to handle real-time events means making it more deterministic so that it can process data reliably inside deadlines. Determinism usually comes at the price of total throughput so a real-time system is not going to be able to process as much data as a non-real-time system.

It is not possible to provide mathematical proof that a complex operating system like Linux will always meet a given deadline, so the only approach is through extensive testing using tools such as `cyclictest` and `Ftrace`, and, more importantly, using your own benchmarks for your own application.

To improve determinism, you need to consider both the application and the kernel. When writing real-time applications, you should follow the guidelines given in this chapter about scheduling, locking, and memory.

The kernel has a large impact on the determinism of your system. Thankfully, there has been a lot of work on this over the years. Enabling kernel preemption is a good first step. If you still find that it is missing deadlines more often than you would like, then you might want to consider the `PREEMPT_RT` kernel patches. They can certainly produce low latencies but the fact that they are not in mainline yet means that you may have problems integrating them with the vendor kernel for your particular board. You may instead, or in addition, need to embark on the exercise of finding the cause of the latencies using `Ftrace` and similar tools.

That brings me to the end of this dissection of embedded Linux. Being an engineer of embedded systems requires a very wide range of skills, which range from a low level knowledge of hardware, how the system bootstrap works and how the kernel interacts with it, to being an excellent system engineer who is able to configure user applications and tune them to work in an efficient manner. All of this has to be done with hardware that is, almost always, only just capable of the task. There is a quotation that sums this up, *An engineer can do for a dollar what anyone else can do for two*. I hope that you will be able to achieve that with the information I have presented during the course of this book.

Bibliography

This learning path has been prepared for you to enable yourself of harnessing the power of embedded Linux and build your own systems. It comprises of the following Packt products:

- *Learning Embedded Linux Using the Yocto Project, Alexandru Vaduva*
- *Embedded Linux Projects Using Yocto Project Cookbook, Alex González*
- *Mastering Embedded Linux Programming, Chris Simmonds*



Thank you for buying Linux: Embedded Development

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at www.packtpub.com.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

