Community Experience Distilled

Getting Started with Cubieboard

Leverage the power of the ARM-based Cubieboard to create amazing projects

Olliver M. Schinagl

[PACKT] open source* PUBLISHING
community experience distilled

Getting Started with Cubieboard

Leverage the power of the ARM-based Cubieboard to create amazing projects

Olliver M. Schinagl

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Getting Started with Cubieboard

First published: December 2014

Production reference: 1121214

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

ISBN 978-1-78328-157-2

www.packtpub.com

Cover image by Mattia Grillo (mattia.grillo.04@gmail.com)

Credits

Author
Olliver M. Schinagl

Reviewers
Praveen Palanisamy
Benjamin Henrion
Emilio López

Acquisition Editor
Llewellyn Rozario

Content Development Editor
Sriram Neelakantan

Technical Editor
Shashank Desai

Copy Editor
Sarang Chari

Project Coordinators
Aboli Ambardekar
Melita Lobo

Proofreaders
Maria Gould
Ameesha Green

Indexer
Hemangini Bari

Production Coordinator
Shantanu N. Zagade

Cover Work
Shantanu N. Zagade

About the Author

Olliver M. Schinagl is Austrian-born and a software developer at heart with a strong interest in electronic engineering. Embedded software is where both his passions come together. Having lived in the Netherlands for most of his life, Olliver is currently working at Ultimaker, a 3D printer manufacturer, where his love for Linux, free and open source software, and embedded development is satisfied. Having worked on open source projects, and as a longtime member of the linux-sunxi community, Olliver has in-depth and hands-on experience with Allwinner-based hardware.

He always had a desire to teach but a stronger desire to work on open source projects and embedded hardware. Thus, when offered the chance to write a book in his spare time, he decided to listen to his inner voice and took the chance to use the printed form to teach.

Having never done any writing except for academic work, this was both a challenge and a great experience. Hopefully, you will appreciate the effort and not only learn from the things brought via this book, but also gain the appetite to work out creative ideas, put the knowledge to good use, and share it with others so they can then benefit from it.

Writing a book costs time, and to understand and support this, I would like to thank my partner in life, Anshariah, who encouraged and cursed those late night writing sessions. Additionally, I would like to thank my parents and all my friends for always being there for me, supporting me, and being proud as parents and friends would be.

Finally, a pledge of gratitude goes out to all the free and open source software and hardware developers and advocates for all the things they make and create, all the things they share, and all the things I have learned from. It is because of them that I am able to write code and text using all the source tools. It is people like you who, in the end, make the world a better place.

About the Reviewers

Praveen Palanisamy is a robotics, computer vision, and embedded system enthusiast, and he is currently pursuing his Master's degree at the Robotics Institute at Carnegie Mellon University. He got his Bachelor's degree in Electrical and Electronics Engineering from Vellore Institute of Technology University, Chennai. Inventing a machine that can walk, run, think, and interact like human beings has always intrigued and fascinated him.

He is an autodidact who learned computer programming. Spurred by his passion for robotics, he learned image processing and computer vision techniques to program and build intelligent robots and embedded systems. He has worked with a series of ARM architecture-based development boards and CPUs, including Dual-core Cortex-A9-based Pandaboard ES, Cortex-A8-based Cubieboard, and Cortex-M3-based Stellaris IDM L-35. He has also worked with 8-bit AVR RISC microcontrollers and Arduino. He has experience in building Linux systems from scratch on embedded platforms. He is working part-time in a computer vision-based start-up named Cladoop. A list of Praveen's projects and demonstrations can be explored at `http://praveenp.com`.

Benjamin Henrion has been hacking embedded devices since 2000 with the OpenAP/LinuxAP distribution running on the first wireless router that runs Linux. He has been an exclusive Linux user since 1996, and he owns an extensive collection of embedded devices running Linux. He has contributed to the development of wireless routing protocols and initiated the Wireless Battle Mesh event, which aims to test those protocols running on routers based on OpenWrt.

Benjamin is also the President of the Foundation for a Free Information Infrastructure e.V. (`http://ffii.org/`), and he has been fighting software patents since 1999, from the beginning of the European debate till now. He has launched several popular campaigns on the Internet, such as the August 2003 and June 2005 web demonstrations against software patents (400,000 signatures), the PublicGeoData campaign for free maps (5,000 signatures), and the campaign against Microsoft Office's standardization at ISO (100,000 signatures); for more information, refer to `http://noooxml.wikidot.com/`.

He currently works for a VoIP company as a systems engineer. His interests lie in computer science, politics, and mountain biking. His personal website is `http://www.zoobab.com/`.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit `www.PacktPub.com`.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`https://www2.packtpub.com/books/subscription/packtlib`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

Over the last few years, ARM chips have become trendy and ubiquitous, ranging from the phone and tablet market to power-efficient server farms. The low cost associated with the chips in conjunction with their powerful features makes them an apt choice for hobbyists and enthusiasts. In addition to offering a ton of connectivity, these chips have been used by several manufacturers on their development boards. The Cubieboard is a type of board with built-in networking and various input and output ports, making it an awesome utility for myriad purposes, such as media centers, robotic projects, home automation, web servers, and home security systems, to mention a few. The Cubieboard is a microcontroller, which provides a whole new set of capabilities with the extensibility of desktop machines but without the bulk or noise.

Low cost, highly expandable, and high performing with a massive, diverse range of uses and applications, the Cubieboard will revolutionize the way we think about computing and programming. With its power-packed attributes and versatility, you can create fun things. There is absolutely no fixed way to develop complex projects; however, this book will give you enough basics of the Cubieboard in a few different realms so that you can dig deeper on your own.

What this book covers

Chapter 1, *Choosing the Right Board*, starts with an overview of various development boards and compares a few popular ones to help you choose a board tailored to your requirements. You will also take a look at the additional hardware and a few extra peripherals that will help you understand the stuff you require for your projects.

Chapter 2, *Getting Started with the Hardware*, helps you with the initial settings before you can try out things with it. After unwrapping the Cubieboard, you will learn the procedure to connect a serial port to the development board and move on to booting up the preinstalled software.

Chapter 3, *Installing an Operating System*, explains the procedure of installing an operating system in addition to installing a fully-functional graphical desktop environment onto a microSD card. It also points out the difference between an OS image and a clean installation, thereafter moving on to installing Fedora in addition to writing the OS image to a microSD card.

Chapter 4, *Manually Installing an Alternative Operating System*, helps you with the process of installing a customized OS on an alternative medium (SATA SSD) in addition to making the destination medium bootable using the command line.

Chapter 5, *Setting Up a Home Server*, explains how the Cubieboard can be used as a home server efficiently in addition to setting up different services to be used in a home environment. You can learn about the procedure of setting up a web server, file server, torrent server, and then summing it up with setting up a personal cloud.

Chapter 6, *Updating the Bootloader and Kernel*, helps you to understand the difference between the various bootloader and kernel types while also assisting you with the process of obtaining and installing a new bootloader or kernel onto an SD card, which will be used as a boot device. Kernels often get updated to newer versions with security fixes or support for new hardware, thereby making it mandatory to know about them when working with many ARM boards, such as the Cubieboard.

Chapter 7, *Compiling the Bootloader and Kernel Using a BSP*, deals with the board support package (BSP), thereby helping you compile the bootloader and kernel from source when some changes are to be made to the source code of the bootloader and kernel. You will learn to use BSP in conjunction with Git and create an easy-to-use, device-specific hardware pack.

Chapter 8, *Blinking Lights and Sensing the World*, starts with explaining basic electronic concepts and moves on to toggling GPIO pins and then make LEDs blink, thereby encouraging you to try out new things as you make your foray into the world of possibilities with the Cubieboard.

Appendix A, *Getting Help and Finding Other Helpful Online Resources*, educates you on the online resources at your disposal due to the vibrant communities and also how to obtain these resources and get help from the community in general.

Appendix B, *Basic Linux Commands Cheatsheet*, is a collection of various Linux commands that form a major part of your workload while using the Cubieboard, thereby helping you get to grips with the technology.

Appendix C, *The FEX Configuration File*, helps you understand the FEX files that are imperative due to the fact that they are used to configure the drivers.

Appendix D, *Troubleshooting the Common Pitfalls*, is a small guide that will be quite handy when faced with errors and hurdles, such as boot failures, stability issues, and errors that pop up while executing commands.

What is needed for this book

Nearly everything covered in this book can be done be on the board. To communicate with the board, a working PC is required with a working USB port to connect a USB to serial 3.3 volt UART adapter. Depending on the operating system used, a terminal emulator such as PuTTY is required. There are two options to compile the sources used in this book, either natively on the development board or via a so-called cross compiler on a regular PC. If a regular PC is used, Linux is required, but this can be run from within a virtual machine. This book was written using only freely available open source software.

Who this book is for

This book is intended for anyone who wants to start working with Allwinner A10, A13, or A20 ARM-based hardware. This can range from hobbyists and developers at home working on a cool project to professionals working on a new ARM-based product with little ARM or little Linux knowledge. No previous Linux knowledge is required but having it certainly makes things a lot easier. Android is not really addressed in this book, and while Android is a common operating system preinstalled on many of these development boards, this book will not cover Android or Android apps.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The `l` command can be used at any time to get an overview of the available types."

Any command-line input or output is written as follows:

```
packt@PacktPublishing:~$ tar xJvf u-boot-sunxi-cubieboard.tar.xz
u-boot-sunxi-cubieboard-20140307T104232-b5bd4c9/
u-boot-sunxi-cubieboard-20140307T104232-b5bd4c9/u-boot-sunxi-with-
spl.bin
u-boot-sunxi-cubieboard-20140307T104232-b5bd4c9/u-boot.bin
u-boot-sunxi-cubieboard-20140307T104232-b5bd4c9/sunxi-spl.bin
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Make sure to check **Serial**."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via 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 on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, 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 from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Downloading the color images of this book

We also provide you a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from: `https://www.packtpub.com/sites/default/files/downloads/1572OS_ColoredImages.pdf`

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 books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, 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 on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

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

Piracy

Piracy of copyright 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

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Choosing the Right Board

It is that time of the year again when there are a few days to spare, and you are anxious to play with one of these new ARM development boards everybody keeps talking about. There are, however, a lot of boards available. With so many choices available, which board do you pick? Choosing the board to start working with can make a difference later on, so this chapter provides an introduction to the various boards and states the major differences between them. While the focus of this book does indeed lie on the Cubieboard family from Cubietech, it might still be prudent to give this chapter some attention for a potential second board. Additionally, this book does apply just as easily to the boards mentioned here.

In this first chapter, we will cover the following topics:

- Why are there so many boards to choose from?
- An overview of various boards
- Highlighting the most popular boards
- Ideas regarding what additional hardware is required

Wading through the forest of available chips and boards

There are many chips and even more boards to choose from when going into ARM development. This chapter also provides a short introduction to various chips and compares them.

A short overview of chips

In the last few years, ARM-based **Systems on Chips** (**SoCs**) have become immensely popular. Compared to the regular x86 Intel-based or AMD-based CPUs, they are much more energy efficient and still perform adequately. They also incorporate a lot of peripherals, such as a **Graphics Processor Unit** (**GPU**), a **Video Accelerator** (**VPU**), an audio controller, various storage controllers, and various buses (I2C and SPI), to name a few things. This immensely reduces the required components on a board. With the reduction in the required components, there are a few obvious advantages, such as reduction in the cost and, consequentially, a much easier design of boards. Thus, many companies with electronic engineers are able to design and manufacture these boards cheaply.

So, there are many boards; does that mean there are also many SoCs? Quite a few actually, but to keep the following list short, only the most popular ones are listed:

- Allwinner's A-series
- Broadcom's BCM-series
- Freescale's i.MX-series
- MediaTek's MT-series
- Rockchip's RK-series
- Samsung's Exynos-series
- NVIDIA's Tegra-series
- Texas Instruments' AM-series and OMAP-series
- Qualcomm's APQ-series and MSM-series

While many of the potential chips are interesting, Allwinner's A-series of SoCs will be the focus of this book. Due to their low price and decent availability, quite a few companies design development boards around these chips and sell them at a low cost. Additionally, the A-series is presently the most open source friendly series of chips available. There is a fully open source bootloader, and nearly all the hardware is supported by open source drivers. Among the A-series of chips, there are a few choices. The following is a list of the most common and most interesting devices:

- **A10**: This is the first chip of the A-series and the best supported one as it has been around for a long time. It is able to communicate with the outside world over I2C, SPI, MMC, NAND, digital and analog video out, analog audio out, SPDIF, I2S, Ethernet MAC, USB, SATA, and HDMI. This chip initially targeted everything, such as phones, tablets, set-top boxes, and mini PC sticks. For its GPU, it features the MALI-400.

- **A10S**: This chip followed the A10; it focused mainly on the PC stick market and left out several parts, such as SATA and analog video in/out, and it has no LCD interface. These parts were left out to reduce the cost of the chip, making it interesting for cheap TV sticks.
- **A13**: This chip was introduced more or less simultaneously with the A10S for primary use in tablets. It lacked SATA, Ethernet MAC, and also HDMI, which reduced the chip's cost even more.
- **A20**: This chip was introduced way after the others, and even was pin-compatible to the A10 with the intend to replace it. As the name hints, the A20 is a dual-core variant of the A10. The ARM cores are slightly different; Cortex-A7 has been used in the A10 instead of Cortex-A8 used previously.
- **A23**: This chip was introduced after the A31 and A31S and is reasonably similar to the A31 in its design. It features a dual-core Cortex-A7 design and is intended to replace the A13. It is mainly intended to be used in tablets.
- **A31**: This chip features four Cortex-A7 cores and generally has all the connections that the A10 has. It is, however, not popular within the community because it features a PowerVR GPU that, until now, has seen no community support at all. Additionally, there are no development boards commonly available for this chip.
- **A31S**: This chip was released slightly after the A31 to solve some issues with the A31. There are no common development boards available.

Choosing the right development board

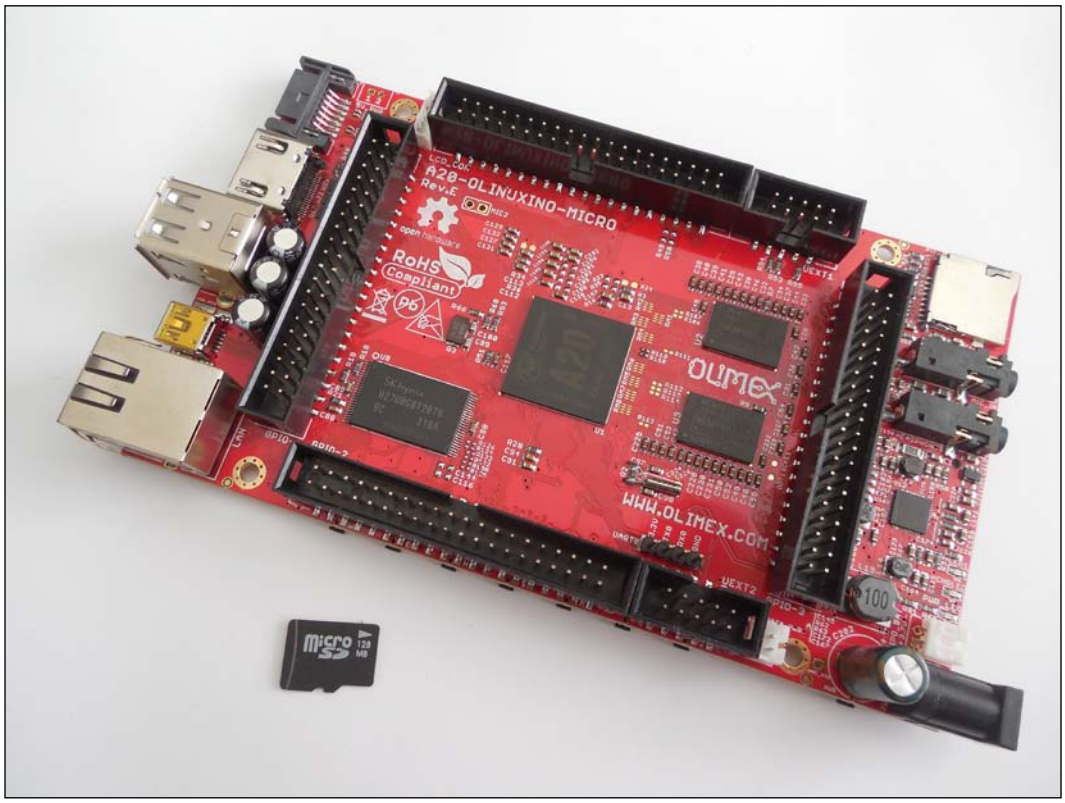
Allwinner's A-series of SoCs was produced and sold so cheaply that many companies used these chips in their products, such as tablets, set-top boxes, and eventually, development boards. Before the availability of development boards, people worked on and with tablets and set-top boxes. The most common and popular boards are from Cubietech and Olimex, in part because both companies handed out development boards to community developers for free.

Olimex

Olimex has released a fair amount of different development boards and peripherals. A lot of its boards are open source hardware with schematics and layout files available, and Olimex is also very open source friendly. You can see the Olimex board in the following image:

Olimex offers the A10-OLinuXino-LIME, an A10-based micro board that is marketed to compete with the famous Raspberry Pi price-wise. Due to its small size, it uses less standard 1.27 mm pitch headers for the pins, but it has nearly all of these pins exposed for use.

You can see the A10-OLinuXino-LIME board in the following image:

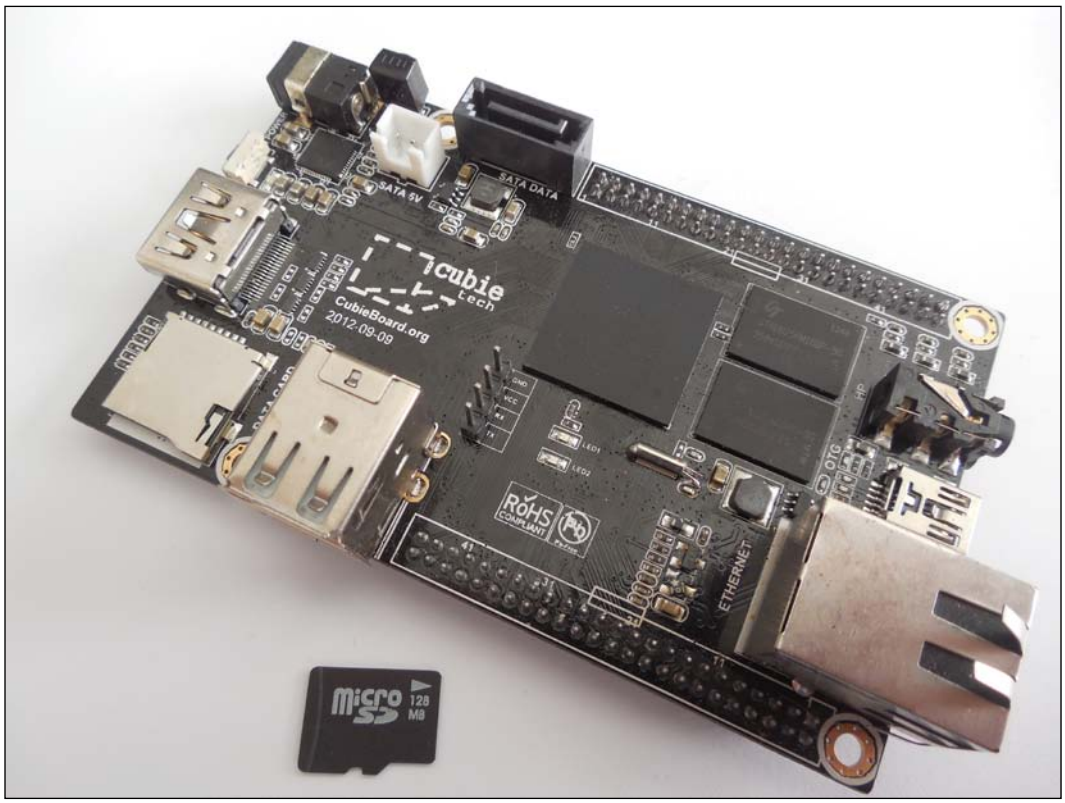


The Olimex OLinuXino series of boards is available in the A10, A13, and A20 flavors and has more standard 2.54 mm pitch headers that are compatible with the old IDE and serial connectors. Olimex has various sensors, displays, and other peripherals that are also compatible with these headers.

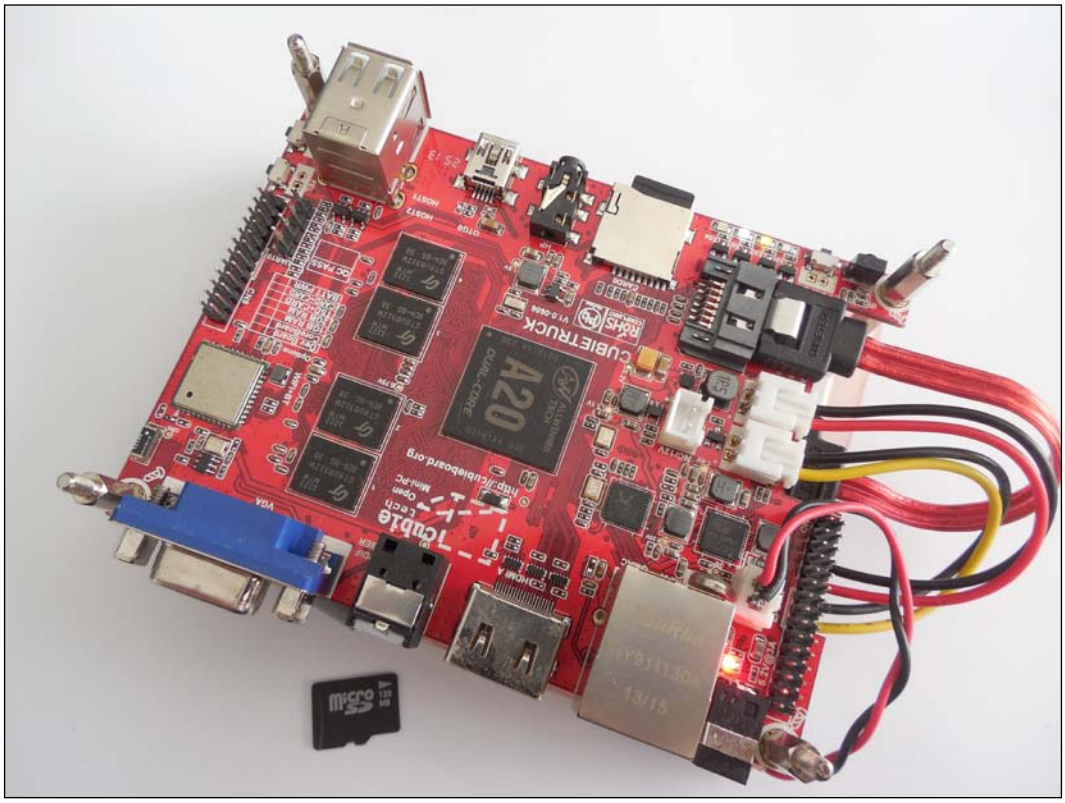
Olimex recently announced that it will be releasing a **System on a Module** (**SoM**). It is identical in concept to the Itead board, which will be mentioned in a later section.

Cubietech

Cubietech was formed by previous Allwinner employees and was one of the first development boards available using the Allwinner SoC. While it is not open source hardware, it does offer the schematics for download. Cubietech released three boards: the Cubieboard1, the Cubieboard2, and the Cubieboard3—also known as the Cubietruck. Interfacing with these boards can be quite tricky as they use 2 mm pitch headers that might be hard to find in Europe or America. You can see the Cubietech board in the following image:



Cubieboard1 and Cubieboard2 use identical boards; the only difference is that A20 is used instead of A10 in Cubieboard2. These boards only have a subset of the pins exposed. You can see the Cubietruck board in the following image:



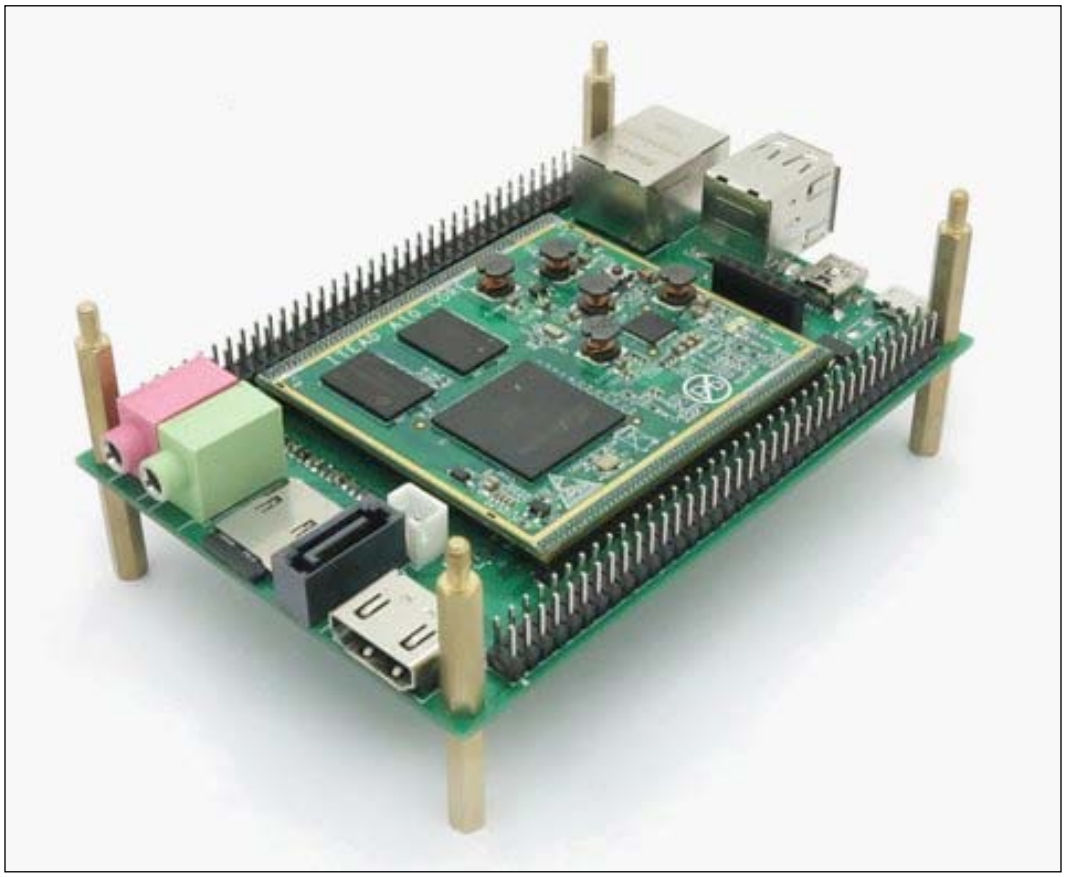
Cubietruck is quite different but is a well-designed A20 board. It features everything that the previous boards offer, along with Gigabit Ethernet, VGA, Bluetooth, Wi-Fi, and an optical audio out. This does come at a cost as there are fewer pins to keep the size reasonably small. Compared to Raspberry Pi or LIME, it is almost double the size.

Lemaker

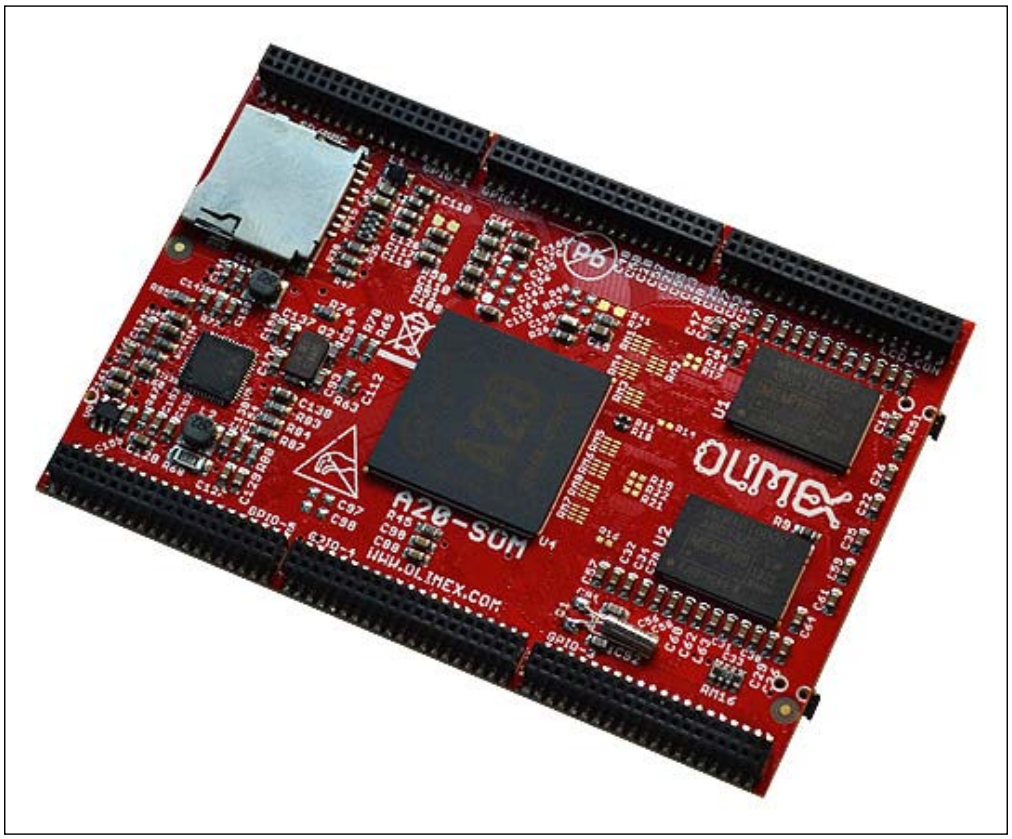
Lemaker made a smart design choice when releasing its Banana Pi board. It is an Allwinner A20-based board but uses the same board size and connector placement as Raspberry Pi, hence the name Banana Pi. Because of this, many of those Raspberry Pi cases could fit the Banana Pi and even shields will fit it. Software-wise, it is quite different and does not work when using Raspberry Pi image files. Nevertheless, it features composite video out, stereo audio out, HDMI out Gigabit Ethernet, two USB ports, one USB OtG port, CSI out and LVDS out, and a handful of pins. Also available are a LiPo battery connector, a SATA connector, and two buttons, but those might not be accessible on a lot of standard cases. See the following image for the topside of the Banana Pi:

Itead and Olimex

Itead and Olimex both offer interesting boards, which are worth mentioning separately. The Iteaduino Plus and the Olimex A20-SoM are quite interesting concepts; the computing module, which is a board with the SoC, memory, and flash, which are plugin modules, and a separate baseboard. Both of them sell a very complete baseboard as open source hardware, but anybody can design their own baseboard and buy the computing module. You can see the following board by Itead:



Refer to the following board by Olimex:



Additional hardware

While a development board is a key ingredient, there are several other items that are also required. A power supply, for example, is not always supplied and does have some considerations. Also, additional hardware is required for the initial communication and to debug.

Serially interfacing with the board

With headless systems such as these, things don't always just work. Sometimes, debugging at a lower level is required. This is also certainly true with development boards such as these. An error occurs, and there's no output; what could have possibly gone wrong? So spending hours on trial and error can be avoided; the old and trusty serial port exists on many types of hardware. With Allwinner's SoCs, they are implemented in two ways.

Universal asynchronous receiver/transmitter

On all of the developer boards discussed throughout this book, there are dedicated pins to connect to the serial port of the chip. If the PC that is used to connect to the developer board has such a serial port, be wary of how this is connected. While both speak the same protocol, they operate at different voltages to do this; thus, a level translator is required at the least. Most PCs are without a serial port these days anyway and have to rely on a USB to **universal asynchronous receiver/transmitter** (**UART**) adapter. When getting a USB to UART adapter, it is important that they are 3.3V TTL. Cubieboards are usually shipped with a USB to UART adapter, as shown in the following image:

The microSD adapter

Sometimes, the UART simply isn't available for use as in the case of most tablets. In such cases, a second UART is made available through the microSD card slot. A specific adapter is required to connect to the previously mentioned UART. In the following image, a microSD to UART adapter can be seen (this specific variant also has the ability to grant access to the JTAG pins):

The microSD card

Usually, a microSD card is the boot medium for these boards. It can be thought of as a bootable CD or USB drive used on a PC. When creating microSD cards to be used as a medium, it is advisable to use a class 10 or faster microSD.

Power supply

Believe it or not, but most developer boards actually come without a power supply; this is usually due to the following reasons:

- Without the power supply, developer boards do not have to pass the FCC regulations
- Importing a power supply might require certain local certification and might be forbidden
- It can give rise to the complexity of the developer not knowing which power supply needs to be sent to a country
- It has reduced the cost, as not bundling the power supply has brought about a reduction in the cost

Most boards will take 5 volts for their input, but 700 milliamp of current is the least that they should supply when not using anything power hungry, such as an HDD or an LCD. If extra peripherals are attached, this requirement also goes up. A proper 5-volt, 2-amp power supply will be enough to power a board under full load with an LCD attached. Depending on the power requirements of the hard drive, even that should be able to work quite nicely. When in doubt, always check the power drain or power supply stability to exclude that from causing strange issues. Cheaply made power supplies available from various shops are often overrated and thus they might not supply the required current; however, they might not supply the required current and cause everything to run unstably.

Summary

After working your way through this chapter, you should now have an idea of the available, popular development boards, and which one might be a good choice for you. Finally, it should be noted that extra hardware is sometimes required or, at least, extremely helpful to work with these boards.

The next chapter will take this newly acquired hardware and explain how you can interact with it. If the selected board comes with preinstalled software, booting it will be covered as well.

2
Getting Started with the Hardware

When you start to play with a new device, there are a few questions that might come to your mind. How do I know it is working properly? How do I get an output from or an input to the device, such as video on a monitor or key strokes from a keyboard? These are probably the first basic questions that are asked after a board is unpacked or started for the first time.

This chapter will cover the following topics:

- Connecting a serial port to the development board
- Booting up the preinstalled software

Connecting a serial port

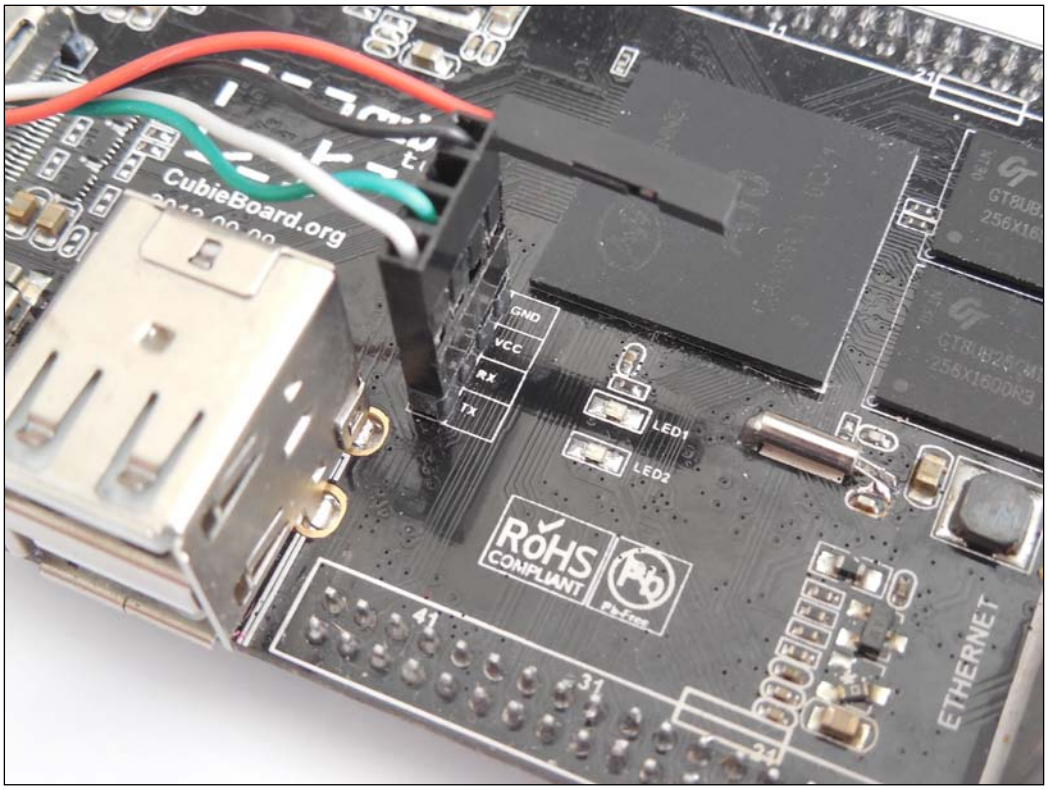
A serial port might seem like something outdated, but it is in fact still very common on certain devices. One of the main reasons is that it is very simple and reliable. It is very simple in both hardware and software implementations, and because of its simplicity, it is often very reliable and pretty much always works—which is why it has been on devices since the sixties until today. A serial connection is, however, rather slow, but for text-based input and output, it is perfectly adequate. However, why would one want a serial port to begin with? As Murphy can attest, things just go wrong, and with these development boards, the serial port is very often the only thing providing any output.

There are two ways to connect the serial port or UART to the device, either with a USB to UART adapter or with a real serial port and a level shifter to convert the voltage to the appropriate voltage.

Since most PCs actually lack a serial port, only the USB to serial approach will be discussed here. The first step is to connect the USB to UART adapter to your Cubieboard, which can be a challenge in itself as there are some USB to UART adapters with three wires, where others have four or even more. Furthermore, the colors of the cables might differ from product to product. Finally, there are 3.3-volt or 5-volt adapters. Check the user manual of the cable to find out which color corresponds to which signal and whether the voltage is 3.3 volts. In the case of the USB to serial adapter that is shipped with the Cubieboards, for example, the following rules apply:

- Black is GND or ground and connects to the GND pin
- Green is TX or transmit and connects to the RX or receive pin on the board
- White is RX or receive and connects to the TX or transmit pin on the board
- Red is VCC or power and should never be connected, or the device might get damaged

Refer to the following image to see the various connections:



A UART connection on Cubieboard1

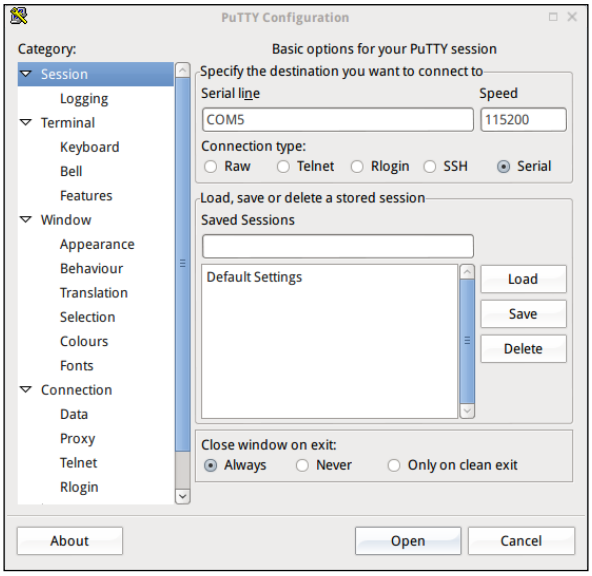
After connecting the UART side of things, the other end can simply be plugged into a USB port. Depending on the operating system used, a driver might need to be installed; furthermore, a program to connect to this serial port is required, and this creates something referred to as a serial terminal. PuTTY is such a program, and it is available on most operating systems. It can be downloaded for the Windows platform at `http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html`.

Other operating systems might have it available via a built-in software store, and it can be installed via this way. Other usable types of software to view the serial console are programs such as GNU Screen or minicom and can be used equally well.

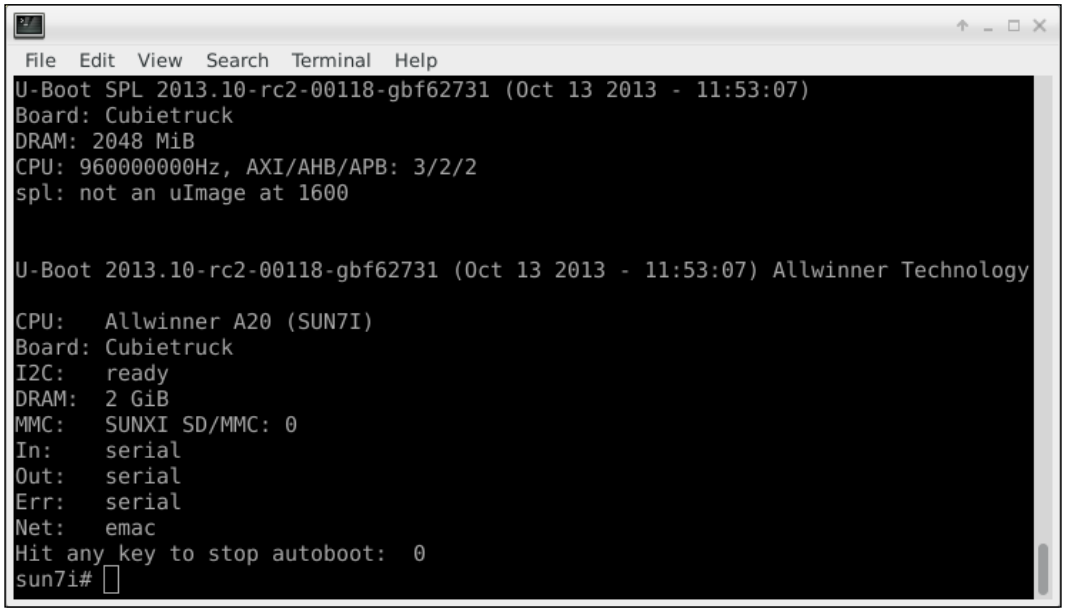
Some parameters are needed to get the serial communication working; a baud rate of 115,200 bits per second is needed. Additionally, eight data bits, no parity, and one stop bit might be needed (which is often abbreviated as 8n1), but both in the aforementioned case of PuTTY or screen (which is the default) might be omitted. For screen, `screen /dev/ttyUSB0 115200` command line can be used, assuming `/dev/ttyUSB0` is the serial port that is being used.

Finding the correct device name or number might be tricky; this not only varies between operating systems, but obtaining the correct number can also be tricky. Under Linux and OSX, for example, using `dmesg` after plugging the USB converter in or using autocomplete on `/dev/ttyUSB` might help. On Windows, the device manager can be used.

For PuTTY, the following screenshot portrays the required settings, assuming `COM5` is the serial port. Make sure to check **Serial**:



If the serial connection was established properly, applying power to the Cubieboard should now yield text on the serial console. An example of this text is shown in the following screenshot. Here, the bootloader that is usually installed on a microSD card is displayed. First, the SPL is loaded, which probes the memory and prints the current CPU configuration, followed by U-Boot, which prints the current configuration. Have a look at the following screenshot:



The preceding screenshot is just an example. This varies between the bootloaders used, the bootmedium, and the board used. It only illustrates what is possibly seen on the first boot.

Booting up the preinstalled software

When a Cubieboard is first powered up, a few things happen. First, the SoC checks various devices to see whether it can boot from them. If available, the onboard NAND flash is very likely to be preprogrammed by the manufacturer. Booting up the Cubieboard using whatever is preinstalled does serve a purpose. It allows you to check whether the Cubieboard is functioning properly.

If the Cubieboard doesn't have a preinstalled operating system or a NAND flash, a specially prepared microSD card can be used. It should yield similar results because the microSD card is actually the first device that the SoC tries to boot. It can be very useful to have one around. The following chapters will give you an idea about how to prepare such an SD card. The preinstalled operating system will very likely require a monitor, keyboard, and mouse connected so that it can be interacted with. While most variants have the Android OS preinstalled, there have been cases where a command-line version of Linux was installed. With an Android preinstallation, various components can easily be tested. Use the following checklist to test the most obvious things:

- Does the display work?
- Does the mouse work?
- Does the keyboard work?
- Does the networking connection work?
- Does the audio playback work?
- Does the MMC card work?

When the display is being tested to see whether it works, it is quite possible that the image is configured to be used with other peripherals than the ones that were expected. For example, an HDMI monitor might be expected, when, in fact, a VGA monitor is connected. The same goes for the audio; it might be routed over to HDMI when a regular headphone is connected via the audio jack.

Going over the aforementioned checklist is harder with a command-line installation but not entirely impossible. The display and keyboard can be tested quite easily. Even the networking feature should be detected. With some basic Linux knowledge, all of the earlier-mentioned components can be easily testable.

For networking to be set up almost automatically, a DHCP server on the network is recommended. Most modems/routers or wireless access points supply this functionality by default.

Summary

Having learned how to connect and use a serial port, you should now be able to watch a Cubieboard boot and use the preinstalled software to check whether things are working normally.

In the next chapter, you will finally start to get some real work done that is, you will set up a full-blown desktop system.

3
Installing an Operating System

Having a Cubieboard is only useful if you can actually use it and eventually develop and/or play with it. The preinstalled OS might or might not be adequate for this. Very often, Android is preinstalled on these devices, as the SoC used is usually found in Android devices, and the manufacturer mostly or only supports it. While BSD or Minix are also operating systems that are being developed by various developers, this book will limit itself to Linux as an operating system. The first few sections of this chapter will dig a little deeper into the concepts for educational purposes.

This chapter will cover the following topics:

- Finding out where the SoC chip decides to boot from
- The difference between an OS image and a clean install
- Downloading and installing Fedora
- Booting the freshly installed OS from an SD card
- The basic concepts on using Fedora and connecting to a wired network
- Maintaining Fedora via the Package Manager

Booting the Cubieboard

While it might seem natural that the system simply boots, there is a lot more to it. The Allwinner series of SoCs has something called a **Boot Read Only Memory** (**BROM**). The BROM is really a small program embedded into the chip itself that always gets executed first. This program has a few drivers for a minimal set of hardware to ensure that it is small and simple.

First, the BROM will try to find a valid bootloader on the first SD card, also called the MMC slot. If nothing is found there, the NAND is checked for a valid bootloader. Again, if nothing is found there, the second MMC slot is checked. If nothing is found there either, the first SPI bus is probed for an SPI memory flash chip and checked for a valid bootloader. Finally, if all of the preceding methods fail, the FEL mode is entered. The FEL mode is a recovery mode where it is possible to upload a piece of code over a USB connection and execute it. This can be useful to recover a board when it fails to boot and the first MMC slot is not available to boot from. The FEL mode does not have to be initiated. One might wonder why two MMC slots are being probed. Sometimes, manufacturers include an embedded MMC chip, or eMMC, in their design, which unlike an SD or MMC card looks like a regular chip, similar to a NAND flash chip, but behaves and looks like an MMC card. By probing the two MMC slots, you can have a board that uses an eMMC chip on the board while still having the first MMC slot available for use cases such as boot recovery. The Cubietruck is available with various combinations of these storage options.

OS image installation background

Many sites and forums on the Internet speak of firmware or ROMs when talking about an installation for an embedded device. It all sounds very mysterious initially, but this is far from the truth. A ROM is nothing more than a full disk image, which, in turn, can be written to storage, which the board can boot from. The name *ROM* is derived from the fact that the data used to be stored in a ROM chip. Many of the OS images that can be found for Allwinner development boards are tailored to a very specific board. This is not surprising, as the chip used might be the same on a range of boards, but certain attached peripherals might be completely different; one board may have different memory chips for example. Another board might not have an onboard flash and rely completely on the secondary MMC slot for its OS. It is almost impossible to have a single disk image that works on all these different combinations of hardware, and thus a clean installation sounds like quite a sensible approach. While distributions are slowly starting to support the ARM-based systems, it is still not easy to have a common installer that can install the distribution to any ARM-based board. To bridge these worlds, a few linux-sunxi community members and Red Hat employees have developed a hybrid installation of Fedora, where a generic image is downloaded, configured, and finally launched in the installer.

Getting and preparing Fedora

There have been several releases of Fedora for the A10, A10S, A13, and A20 series of SoCs. As this book focuses on the most recent version, Fedora 20-r1, the latest version is recommended. Additionally, if fails, Fedora 20-r1 can be written to an SD card, which can be used as a recovery boot disk. The first step in this endeavor is to download this disk image. About 2 GB of free disk space is required.

Fedora for this chapter can be downloaded from the Packt Publishing website in the **Help & Support** section of the book's **Support** page.

The next few steps assume that there is a Linux computer available. If this is not the case, the preinstalled OS on the Cubieboard can also be used; while a little bit trickier, it should be quite possible. If the preinstalled OS is Android, then a terminal application will be required that might not be installed. Do note that on Mac OSX, the device path names will be different. Finally, a virtual machine can also be used, but a detailed explanation of that is out of the scope of this book.

Mac OSX uses device nodes that are similar to Linux but slightly different. For example, the second partition on an inserted USB stick can be called at `/dev/sdb2` on Linux, where on OSX however, this would translate to `/dev/disk2s2`.

Writing the OS image to the SD card

First, the image needs to be written to a microSD card that is at least 4 GiB in size.

The next few steps will delete all the content on the SD card.

The microSD card should be connected to the PC. If there is no card reader available, a USB to microSD card reader can be used instead. The `xzcat` command is used to decompress the downloaded `xz-compressed` archive onto the SD card.

In the following example, it is assumed that the microSD card is inserted into a USB card reader and has been assigned the device node, `/dev/sdd`. It is up to the reader to determine the proper device node on the system, but `dmesg` or one of the installed graphical disk utilities can provide an answer here. An example output with the SD card found on the device node `/dev/sdd` is as follows:

```
usb 2-5: new high-speed USB device number 14 using ehci-pci
usb 2-5: New USB device found, idVendor=14cd, idProduct=8123
usb 2-5: New USB device strings: Mfr=1, Product=3, SerialNumber=2
```

```
usb 2-5: Product: USB 2.0  SD MMC READER
usb 2-5: Manufacturer: SDMMC MA8123
usb 2-5: SerialNumber: 312811122181
usb-storage 2-5:1.0: USB Mass Storage device detected
scsi14 : usb-storage 2-5:1.0
scsi 14:0:0:0: Direct-Access     USB 2.0  SD MMC Reader         PQ: 0
ANSI: 0 CCS
sd 14:0:0:0: [sdd] 248320 512-byte logical blocks: (127 MB/121 MiB)
sd 14:0:0:0: [sdd] Write Protect is off
sd 14:0:0:0: [sdd] Mode Sense: 03 00 00 00
sd 14:0:0:0: [sdd] No Caching mode page found
sd 14:0:0:0: [sdd] Assuming drive cache: write through
sd 14:0:0:0: [sdd] No Caching mode page found
sd 14:0:0:0: [sdd] Assuming drive cache: write through
 sdd: sdd1
sd 14:0:0:0: [sdd] No Caching mode page found
sd 14:0:0:0: [sdd] Assuming drive cache: write through
sd 14:0:0:0: [sdd] Attached SCSI removable disk
```

This command might require root privileges; to do so, prefix `dmesg` with `sudo`. The filename used here should match the file downloaded. A cache-flush via the `sync` command is forced upon successful completion. Flushing the cache is important so that we know all the data that has actually been written to the SD card and it is not held up in the cache. This process might take quite a while—10 minutes is commonly reported commonly reported. The following command is an example of writing the image to an SD card and flushing the cache:

```
root@packt:~# xzcat Fedora-Xfce-armhfp-20-a10-1-sda.img.xz > /dev/sdd
&& sync
```

Writing the OS image should be possible on Linux, OSX, the BSDs, Solaris, and many modern POSIX-based systems. On Windows, a little more care is required. A program such as 7-Zip can be used to decompress the image, and an image writer such as WinDD can be used instead.

When done, remove the USB device and reinsert it to force re-reading of the SD card's partition table. Opening a file manager will, in most cases, show the newly created partitions on the SD card, one named `u-boot` and the other `rootfs`.

Writing the bootloader

The bootloader is very device-specific and can even vary between production runs of the same board. This is because the memory initialization is performed by the bootloader and thus can be different. To do this, a setup script is preinstalled in addition to several bootloaders and several kernels for the various generations of SoCs. This script needs to be executed from the partition labeled `u-boot` on the microSD card. If the currently running OS mounts the partitions automatically, the `mount` command can be used to find the mount point, as follows:

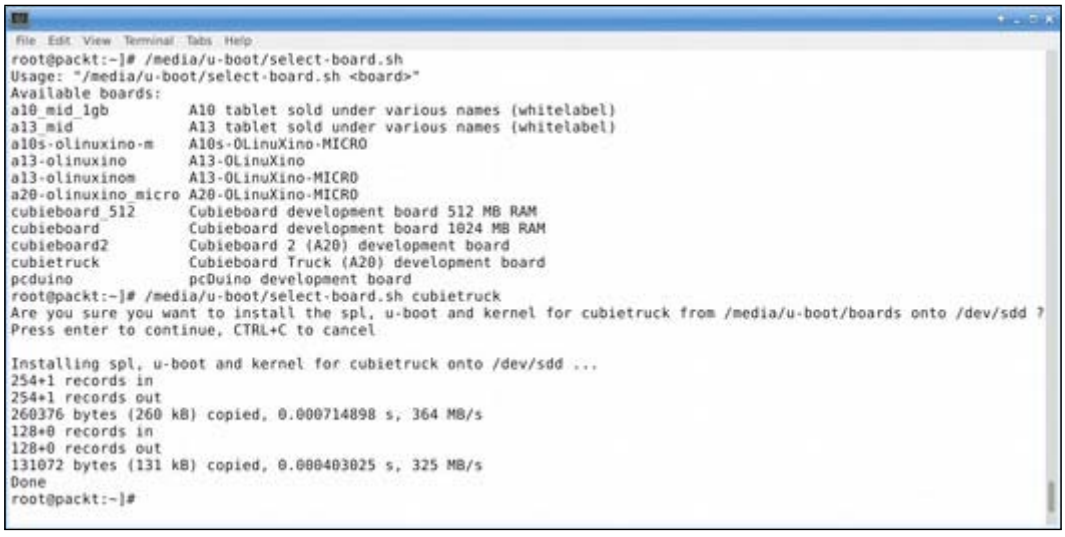
```
[root@packt:~]# mount
/dev/sdd1 on /media/u-boot type ext2 (rw,errors=remount-ro)
```

In the preceding example, the microSD card is mounted on `/media/u-boot`, and the setup script should be run from there. This path should be adjusted as needed. The following is an example showing a sample output of the available boards. The bootloader is being installed using the Cubietruck. It may be required to prefix the command with `bash` to force `bash` to execute the script.

```
[root@packt:~]# bash /media/u-boot/select-board.sh
```

If no Linux system is available, the setup script should be able to run from within the native Android. However, an ADB or a terminal application will be required.

This will now provide a list of supported boards. Find the exact board being used, and run the command again with the selected board as the parameter. Note that only a small selection will be displayed, as shown in the following screenshot:



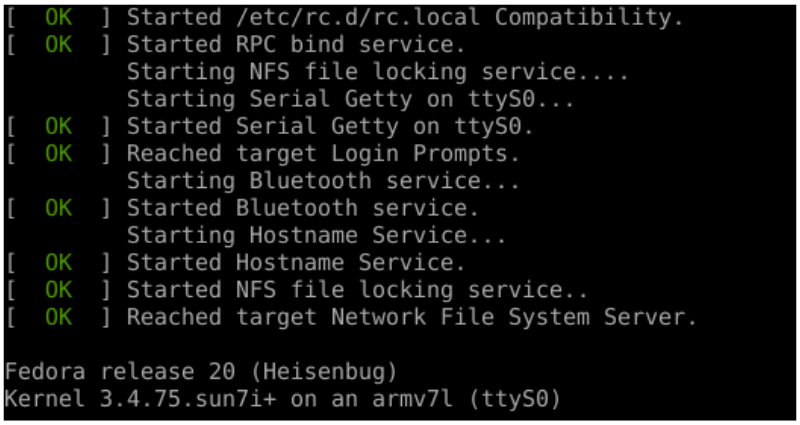
After setting up the board, unmount the microSD card followed by a sync to ensure all the data is written properly. Let us take a look at the following command:

```
[root@packt:~]# umount /dev/sdd1 && eject /dev/sdd1 && sync
```

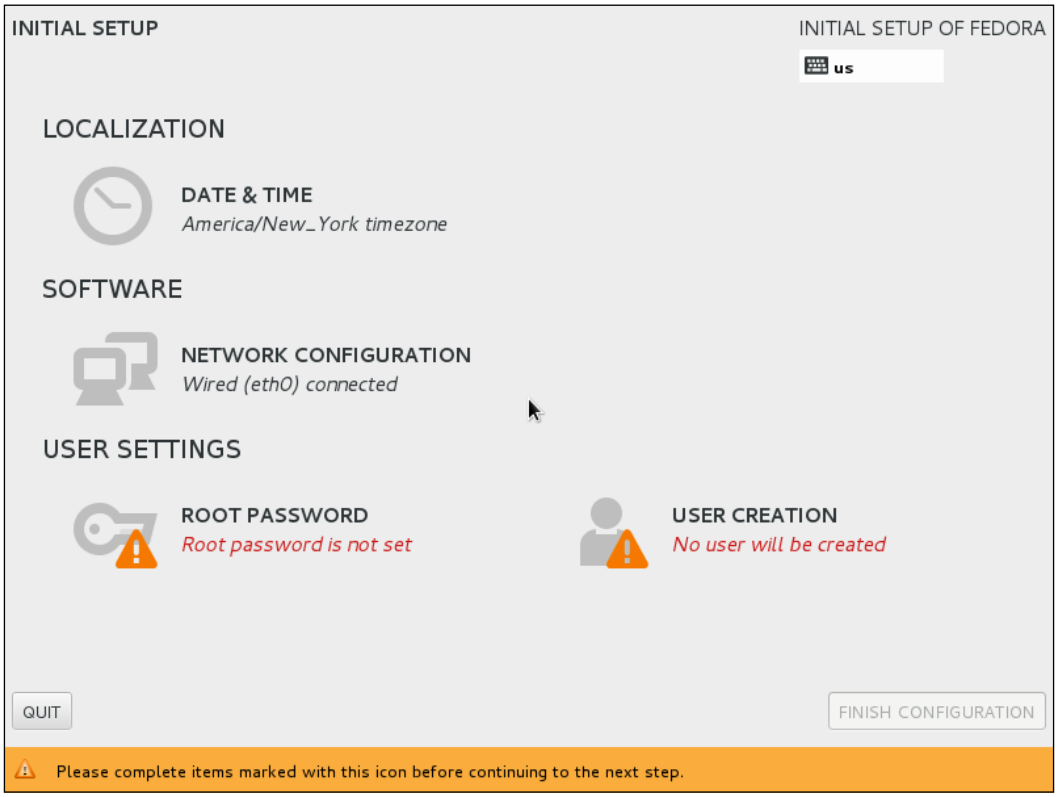
Depending on the environment that this script is being run on, a graphical version can be launched instead; the idea, however, is identical: choose the correct board and the script will write the image to the correct place.

Finishing the operating system installation

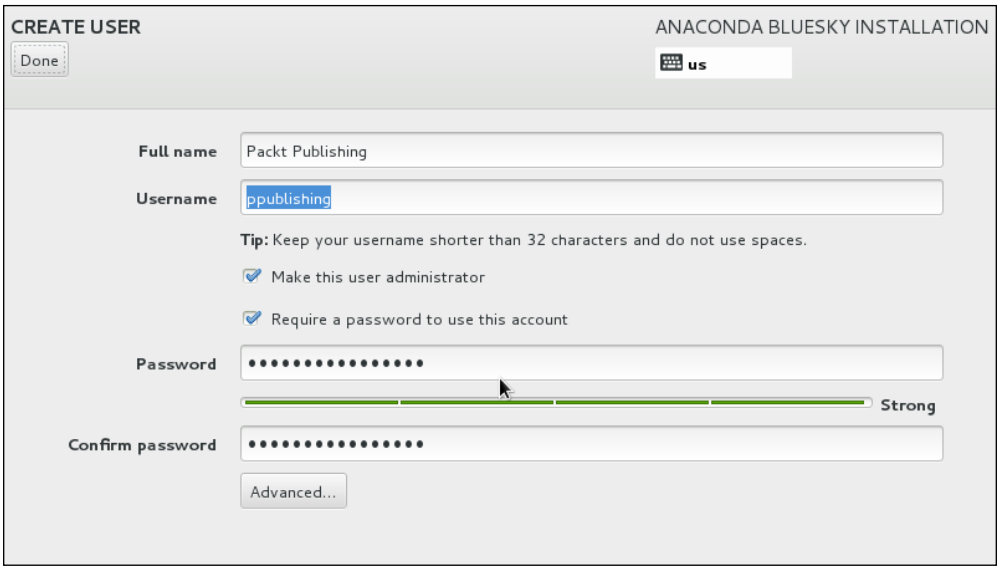
Connect a monitor, USB keyboard, and USB mouse, and insert the microSD card into the Cubieboard. Each board has a default output configured upon the first boot. For many headless boards, this will be the HDMI port. For tablets or systems with LCD screens, it will be the LCD screen. While not strictly required, connecting the UART, as mentioned in the previous chapter, can be helpful in case things go wrong, as shown in the following screenshot:



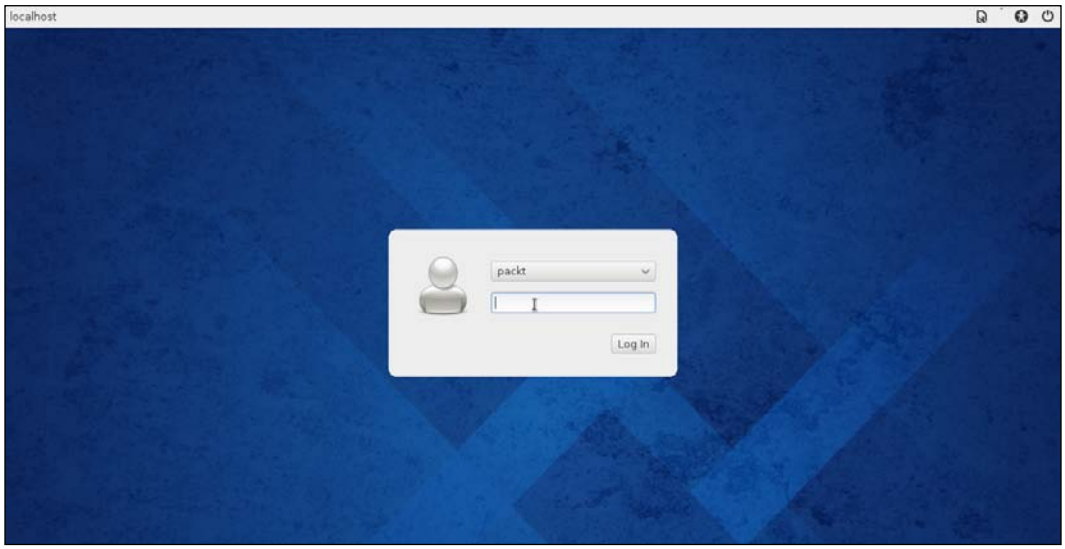
Applying power will boot the device, and after a few minutes and a few intentional reboots to resize the partition on the SD card, the Fedora installer should pop up. If there is no output on the monitor, refer to *Appendix D*, *Troubleshooting the Common Pitfalls*. Also, refer to the following screenshot to see Fedora's first graphical installer screen:



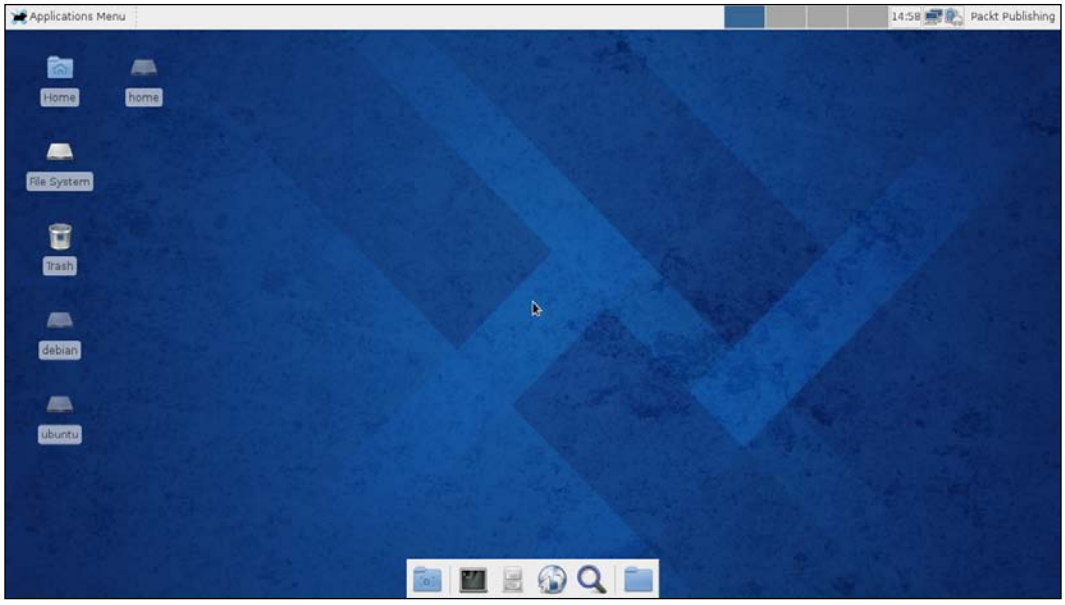
If a mouse or, at least, a keyboard is connected, various parameters for the system can be set up. Verify that at least the time zone is correct, and set up a password for the root user. Also, a new user should be created for regular use of the system, as shown in the following screenshot:



After creating a user and finishing the installation and a short reboot, a login screen should appear, allowing the newly created user to log in. It is now possible to log in to the desktop, as you can see in the following screenshot:



The desktop environment that will be encountered is called **Xfce4**, where the letters only have a historic meaning, and the number indicates the version. Version 4 has been in active development since 2003. Xfce4 is a very lightweight desktop environment and is the default for Fedora on ARM to keep the strain light on the available system resources. Xfce4 might ask the new user a question or two on how the desktop should appear, but the default settings should work fine and were used in this example as well. Feel free to explore this new desktop, launch some applications, or just browse the Internet. Refer to the following screenshot to see the default desktop:



Precautionary measures for installing updates

While there is a perfectly usable desktop environment now, one of the very common tasks is to keep the OS up-to-date and thus secure. Fedora comes with a graphical frontend to the Yum command-line tool called **Yum Extender**. Before going there, however, a warning needs to be issued. At this point, Fedora, like all other distributions, does not officially support the Allwinner range of SoCs. This has one major drawback when updating the OS. The updater, be it Yum or Yum Extender, will also try to update the kernel and bootloader configuration.

At the time of writing this book, the Fedora 20-r1 release will try to update the kernel and bootloader configuration, causing an unbootable device. However, it can easily be fixed by inserting the SD card into a working system and running the `select-device.sh` script as before. To prevent this corruption, edit the file at the `/etc/yum.conf` location, and add the following line to that file, which will force Yum to ignore any kernel updates:

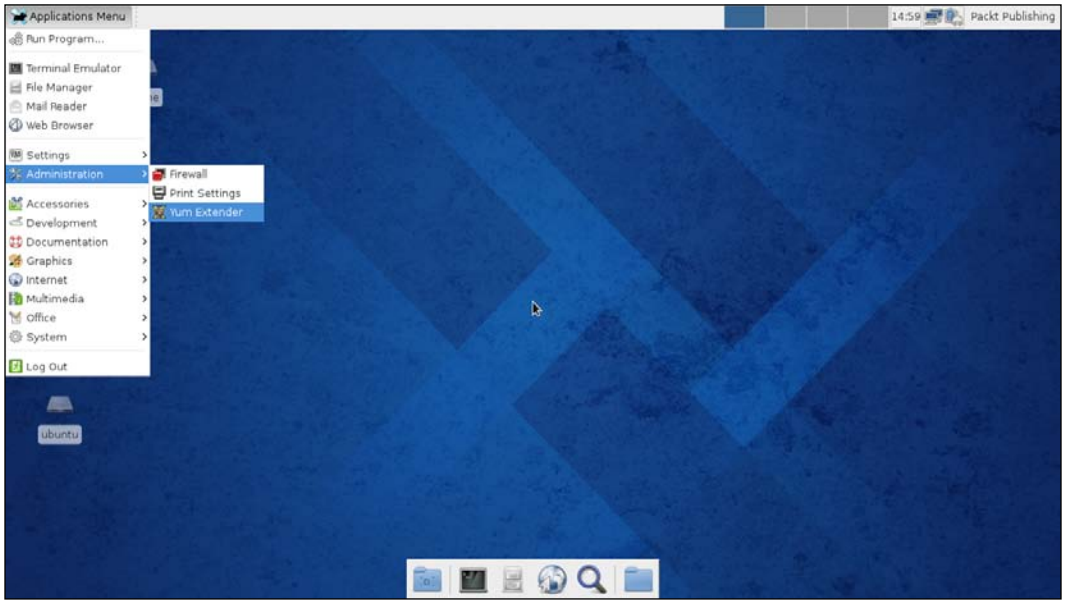
```
exclude kernel*
```

The kernel exclusion should be added to any recent release of Fedora because it might not be applicable to the later versions.

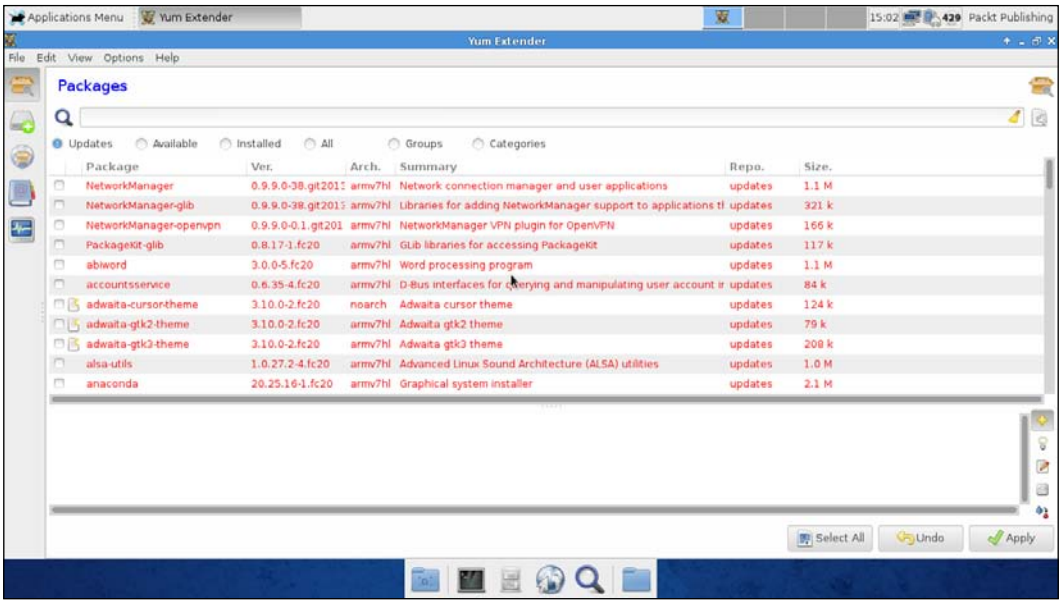
Since the kernel and bootloader live in their own partition, not having the `u-boot` partition mounted can avoid the need to update the bootloader and kernel. In the next chapter, the reader will be able to edit his/her `fstab` file without any difficulty.

Maintaining the OS and installing updates

As mentioned before, the Yum Extender can be used conveniently from the GUI to update the system or install new packages. The Yum Extender can be found under the **Administration** tab in **Applications Menu**, as shown in the following screenshot:



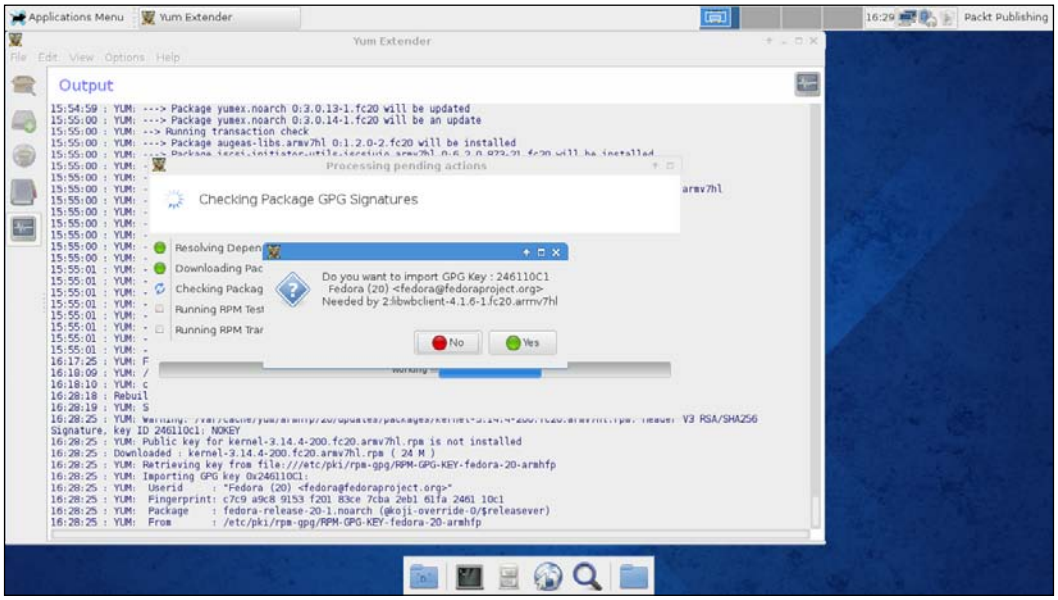
As the Yum Extender is an administrator's application, the current user will have to authorize its use. Assuming this, the user will be granted administrative privileges. The first user of the system will have these privileges. Refer to the following screenshot for a view of **Yum Extender**:



Clicking on the **Select All** button followed by **Apply** will start installing all the new updates to the system. Depending on the `speed` class of the SD card, this can take a long time. While preparing the illustrations for this chapter on an 8 GB class 4 SD card, the process took 5 minutes short of 6 hours. Thus, it is recommended to use at least a class 10 or a better SD card. While using the OS, a class 4 card, even though slow, is usable however.

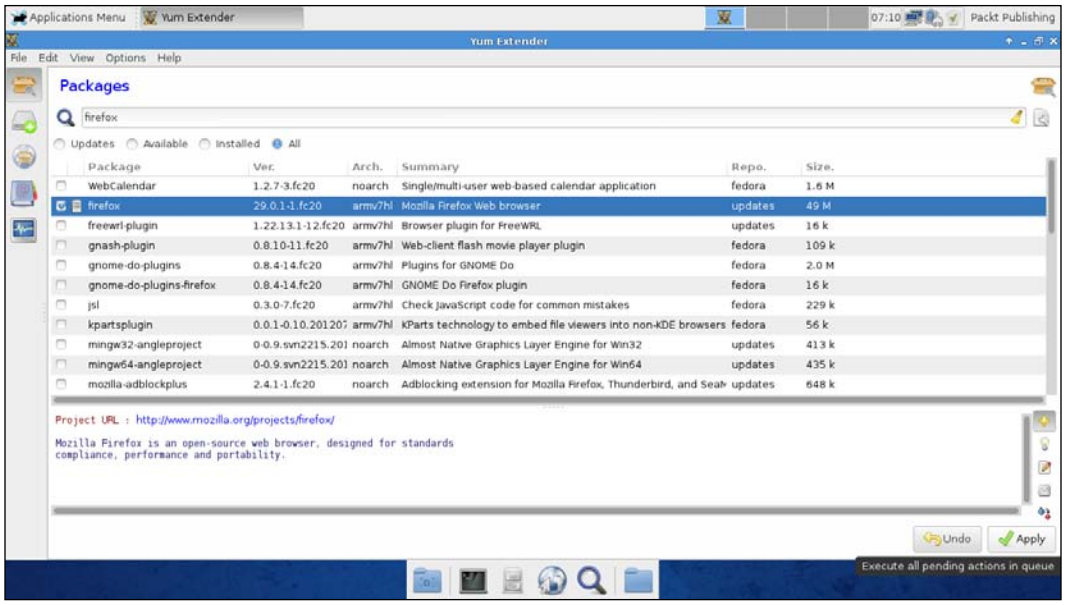
It might be wise to hold out updating the SD card for a Friday evening, for example, where the process can take all weekend without bothering anybody.

If a warning, as illustrated in the following screenshot, appears during the update process, it is safe to accept it. It is a notification to import a missing GPG key. In this case, it was sanctioned by the Fedora project as can be seen in the e-mail address used. This message may come up early in the update process, in the first 10 minutes or so.



Adding more software to the OS

Installing additional software is very easy thanks to Package Managers that have been present in Linux distributions for years. The Yum Extender can not only be used to update the OS, but also to add new software to the system, as seen in the following screenshot, which demonstrates Firefox being installed. Optionally, the command-line version, yum, can be used instead.



Summary

Having gone through this chapter, you should have no problem creating a fresh SD card with a full Xfce4-based Fedora operating system. Also, keeping it up-to-date and adding new software should not be a problem at all. All thanks to a hybrid-disk image of Fedora!

The next chapter will do an installation in a more in-depth fashion manually and focus more on a command line-based interface, as this is often used on server setups, embedded systems, and so on. An entire installation will be performed manually from scratch to learn how to create a customized system.

4

Manually Installing an Alternative Operating System

Installing a full-blown desktop **operating system** (**OS**) onto an SD card via an image is certainly useful, but limitations arise quickly. What if an installation to an SSD is desired? Or what about having a very minimal installation for use as a server? Surely no heavy GUI is needed for this? All these questions will be covered in this chapter.

This chapter will cover the following topics:

- Partitioning and formatting a destination medium
- Creating rootfs
- Allowing booting of the destination medium
- Updating the OS
- Installing additional software

Prerequisites for this chapter

In this chapter, Debian (or even Ubuntu) will be installed to an alternative installation medium. A SATA SSD will be used, but a regular SATA disk can be equally used given that enough electrical power is available to power the drive; a power adapter with at least 2 amperes is required in such a case. Alternatively, a second microSD card can be used in a microSD-to-USB adapter.

When installing to a SATA drive, however, the Cubieboard still requires an SD card to boot, as the SoC cannot boot from a SATA drive directly. Technically, the onboard NAND flash or an onboard SPI flash could be used for this instead, but SPI flash-enabled Cubieboards are hard to find, and working with NAND requires a very old u-boot, which lacks a lot of new features. In this chapter, the microSD card created in the previous chapter will be repurposed and used to accomplish all these tasks.

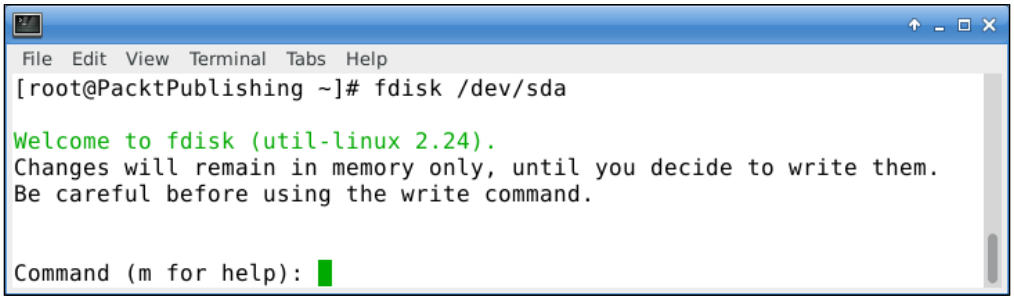
USB flash drives can, in theory, be used, but at the time of writing this book, the USB boot code has not landed in u-boot. At the moment, a USB flash drive can only be used after booting a kernel with USB support, which has been loaded from either a NAND or an SD card.

Preparing the destination medium

To install Debian to a SATA drive, the destination drive will need some preparation. It needs to be partitioned and formatted.

Assuming the Cubieboard is booted up using the Fedora image created earlier and has either a SATA drive, USB flash drive, or a microSD card with a USB adapter connected, it is time to start `fdisk` on the destination medium, which is assumed to be `/dev/sda`. Please make sure that the correct device node is used; otherwise, the following actions will destroy anything present on the medium.

The most common tool of choice to partition a device is `fdisk`. While `fdisk` has a few parameters, starting it with a device node will start `fdisk` in an interactive mode where a disk can be prepared. Root access is required to use `fdisk` and thus may need to be prefixed with `sudo`, as shown in the following screenshot:



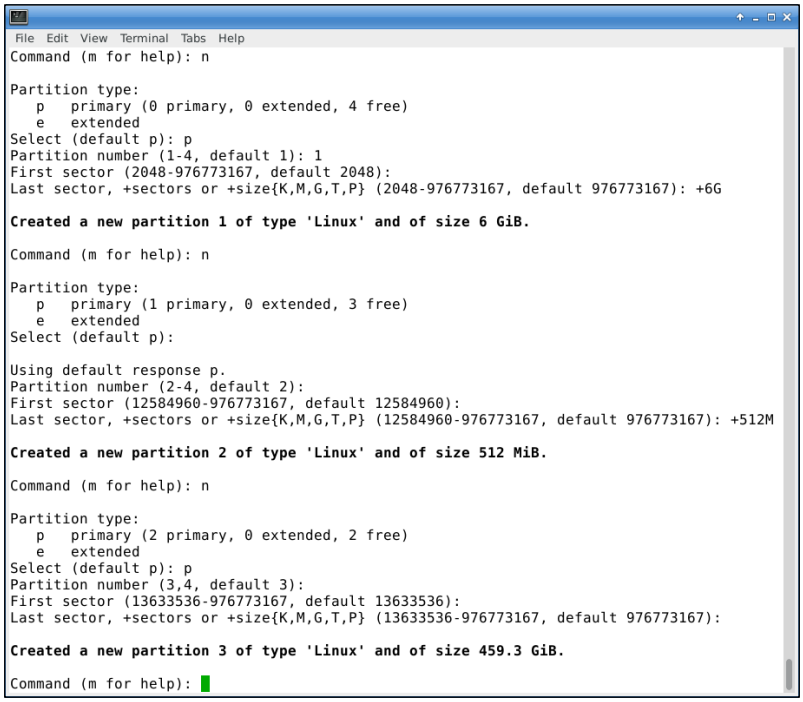
There should not be any previous data present or at least be backed up, which will not be covered here. Pressing the *o* key should clear any previously created partitions, as shown here:

Partitions are a useful thing, they allow a logical separation. There are many reasons and choices when dividing a disk, as but here, only the following four partitions are of interest:

- **Boot**: This partition holds all the relevant boot files
- **Root**: This partition holds all the relevant system files
- **Home**: This partition holds all the user files
- **Swap**: This partition expands the RAM memory

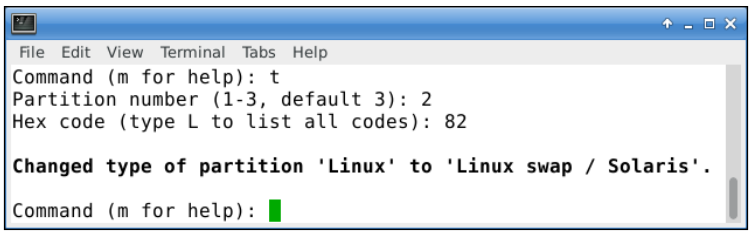
In this example, only three primary partitions will be created as the boot partition will be on the SD card. For the root partition, 6 GB is used, and for a swap partition, 512 MB is used. The remainder is used for the home partition.

Ultimately, however, it is up to the reader to define what is useful to you as this can differ greatly. With the `n` command, a new partition can be created. In the following screenshot, the three partitions explained earlier are created:

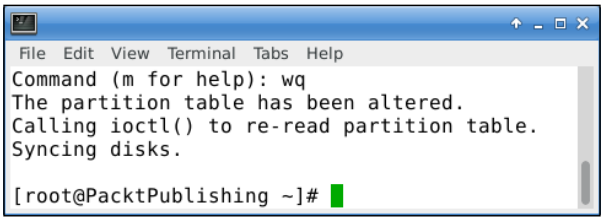


When not entering a value for the last sector, `fdisk` will use the last sector available as default, thus using the remainder of the medium.

Partitions should not only be created, but also need to be categorized. By default, `fdisk` will turn all newly created partitions into regular Linux filesystem partitions, which is fine except for the swap partition. This partition needs a different type applied to it. The `t` command is used to categorize a partition, which in turn wants to know the exact type to use. For swap, this is type 82. The `l` command can be used at any time to get an overview of the available types. The following screenshot shows how to turn partition `2` into a swap partition:



Using the commands `w` to write and `q` to quit, the newly created partition table is saved to the disk and `fdisk` quits, as shown here:



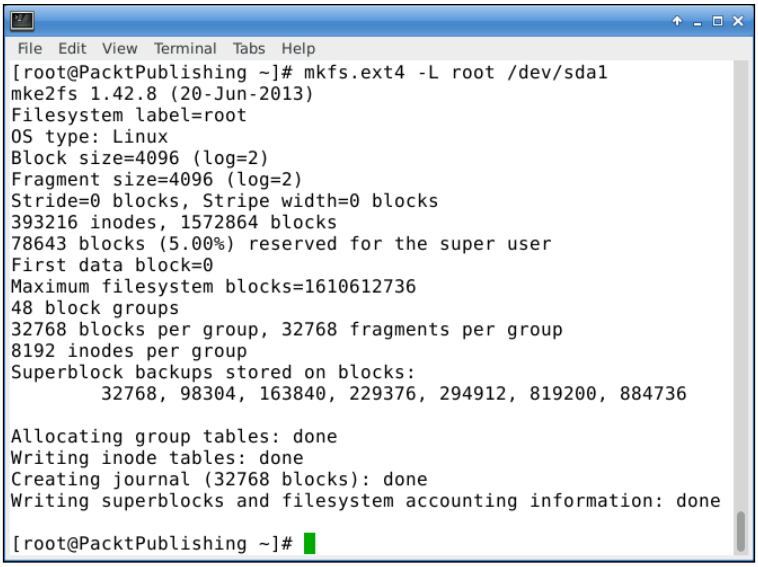
Formatting the newly created partitions

With freshly created partitions available, they now need to be formatted. In this book, `ext4` will be used as the filesystem.

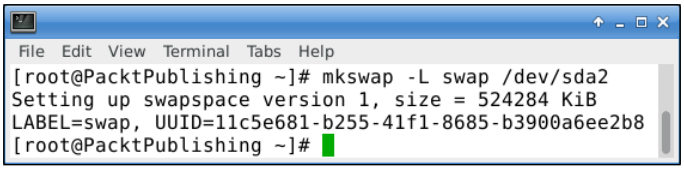
While not yet supported by the 3.4.x kernel used in this book, `f2fs` is very interesting as it is optimized for SSD, USB, or microSD flash usage. This is something that could be interesting in future.

The command to format a partition is `mkfs.ext4`, and the parameters that are of interest are the device node being formatted and optionally `-L`, which is used to give the partition a name.

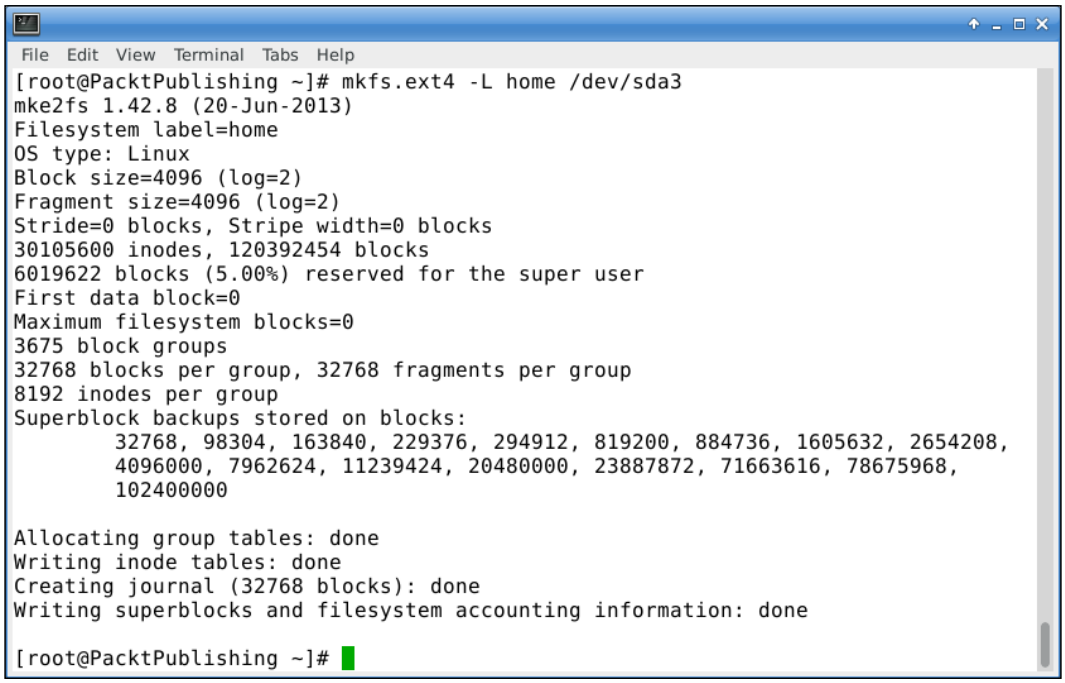
Formatting the root partition is shown here:



Formatting the swap partition can be done using the command in the following screenshot:



Formatting the remainder for the user files can be done as shown here:



To ensure that all the data is written to the appropriate places, the partitions are mounted on the existing filesystem as follows:

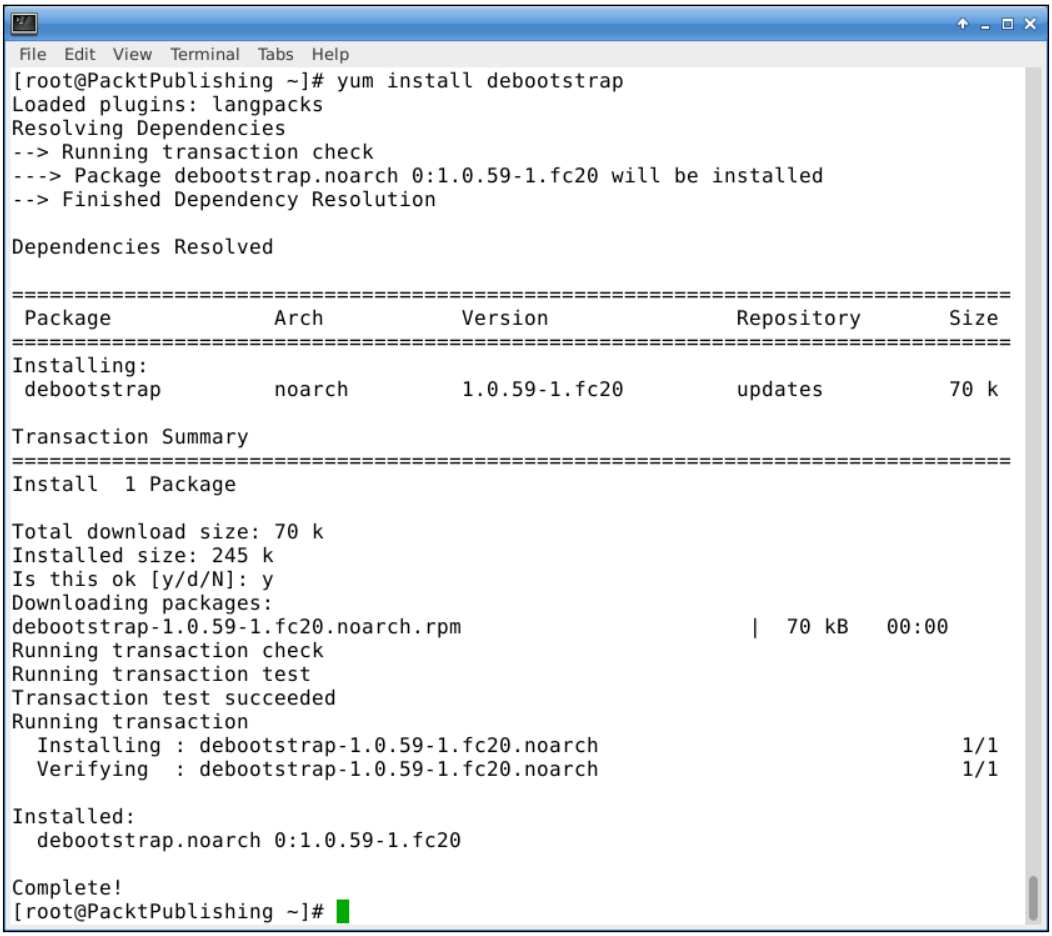
```
[root@packt ~]# mount /dev/sda1 /mnt
[root@packt ~]# mkdir /mnt/home
[root@packt ~]# mount /dev/sda3 /mnt/home
```

Creating a Debian or Ubuntu rootfs

The first thing that should be mentioned here is that Ubuntu is a Debian derivative. To cut a long story short, it is basically Debian, and as a result, it does not hugely matter which distribution is used for installation, be it Debian or Ubuntu. So when talking about installing Ubuntu or Debian, the same thing is really being said. The tool used for installation here is called **debootstrap** and is available in a lot of distributions.

Installing debootstrap

Fedora has debootstrap available and can be installed via the Yum tool, as shown here:



Running debootstrap

With debootstrap installed, it is almost time to get started. A few things need to be mentioned first; debootstrap, which stands for Debian bootstrap, can be used to install any Debian variant for any architecture for at least Debian and Ubuntu. It does require a mirror to be supplied to its list of arguments. The list of mirrors can be obtained for Debian at `http://www.debian.org/mirror/list-full`, but for Ubuntu, there is no official list of mirrors. However, using the country code in the URL can result in a mirror, for example, in the Netherlands, `http://nl.ports.ubuntu.com` is a valid mirror. Using a mirror has the obvious advantage that the download will proceed much faster.

Since Allwinner SoCs are based on the ARMv7 architecture, for the architecture, **armhf** will be used for that.

The **suite**, as it is called, depends on what is desired. For Debian, there are the stable, testing, and unstable suites, where **Wheezy** is the name of the stable branch, **Jessie** is testing, and **sid** is the unstable branch. It should be noted that in future, the names Wheezy and Jessie will change to new suite names, but sid will always remain the unstable development version.

Finally, debootstrap is prefixed with the `PATH` variable to ensure debootstrap uses the correct path. This is due to a bug currently in debootstrap in combination with the newer distributions.

The `–foreign` parameter can be used to bootstrap any architecture, even on an x86-based system, as no code is executed. Bootstrapping will require some additional work using the `–second-stage` parameter. It is up to the reader to learn more about this when cross bootstrapping.

The following command will start the installation of Debian Wheezy for arm-hard-float into `/mnt` using the `http://ftp.nl.debian.org/debian/` mirror, as shown here:

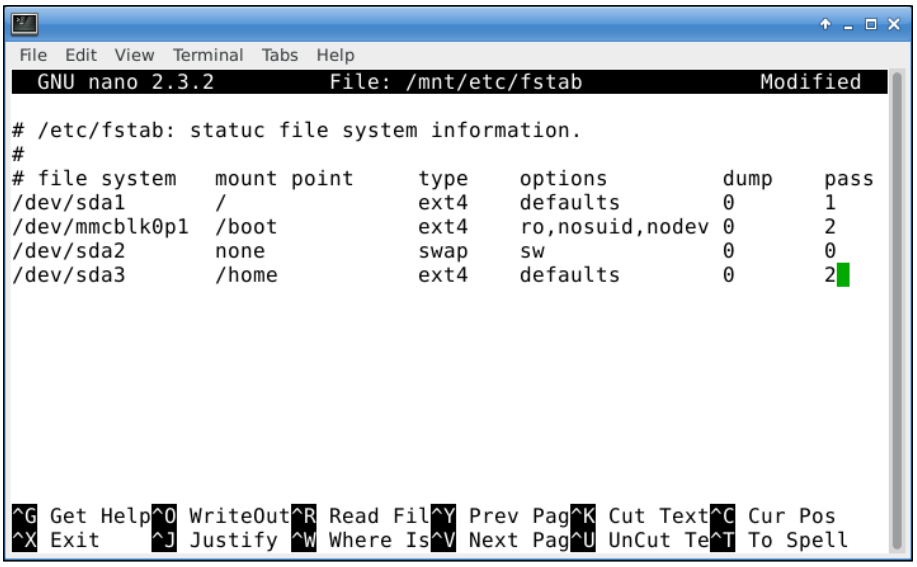
```
File  Edit  View  Terminal  Tabs  Help
[root@localhost ~]# PATH=/bin:/sbin:/usr/bin:/usr/sbin debootstrap --arch=armhf wheezy /mnt http://ftp.nl.debian.org/debian
W: Cannot check Release signature; keyring file not available /usr/share/keyrings/debian-archive-keyring.gpg
I: Retrieving Release
I: Retrieving Packages
I: Validating Packages
I: Resolving dependencies of required packages...
I: Resolving dependencies of base packages...
I: Found additional required dependencies: insserv libbz2-1.0 libdb5.1 libsemanage-common libsemanage1 libslang2 libustr-1.0-1
I: Found additional base dependencies: libept1.4.12 libgcrypt11 libgnutls26 libgpg-error0 libidn11 libnfnetlink0 libp11-kit0 libsqlite3-0 libtasn1-3 libxapian22
I: Checking component main on http://ftp.nl.debian.org/debian...
I: Retrieving libacl1 2.2.51-8
I: Validating libacl1 2.2.51-8
I: Retrieving adduser 3.113+nmu3
I: Validating adduser 3.113+nmu3
I: Retrieving apt 0.9.7.9+deb7u2
I: Validating apt 0.9.7.9+deb7u2
I: Retrieving apt-utils 0.9.7.9+deb7u2
I: Validating apt-utils 0.9.7.9+deb7u2
I: Retrieving libapt-inst1.5 0.9.7.9+deb7u2
I: Validating libapt-inst1.5 0.9.7.9+deb7u2
```

As debootstrap is written in Perl. It may be possible that **Perl** is not currently installed on the system where debootstrap is being used. Installing Perl may follow a long list of packages that need to be downloaded and installed.

Similarly, debootstrap `–foreign --arch=armhf trusty /mnt http://nl.ports.ubuntu.com` can be used to install **Ubuntu Trusty Tahr** with a note that this will only be the base system. Also, debootstrap might not come with all the suites as expected. All Ubuntu suites are symlinks to the gutsy suite at `/usr/share/debootstrap/scripts` as long as it exists on `http://ports.ubuntu.com/dists/`. Use `ln -s gutsy utopic` in the scripts directory, for example, to add utopic as a valid suite.

Configuring the base system

The so-called `fstab` file in Linux is responsible for mounting partitions in their designated positions. Using any editor, the following changes need to be added to the `fstab` file. While the UUID-based mount points can be used, only standard entries are used here. However, you are welcome and even encouraged to use the UUID-based mount points. A common editor that is relatively easy to use is **nano**. After modifying the file, exit nano with the *Ctrl* key in combination with the *x* key and answer the question to save the modified buffer with the *y* key. The filename should remain the same, thus answering with the *Enter* key. The `fstab` file in the editor is shown in the following screenshot:



With `/boot` mounted read-only, it makes sure that not only no accidental writes happen, but also no intended writes occur. Also, as mentioned earlier in this chapter, the boot partition of the microSD card, which was created in the previous chapter, is reused here.

Configuring the networking

Debian and Ubuntu use the `interfaces` file at `/etc/network/interfaces` to configure networking. Note that this is a more permanent configuration used when not using the graphical utilities, such as network manager. If the final goal of this setup is a graphical desktop, it is probably wise to skip setting up the `interfaces` file.

Using the network interface `eth0` as the parameter for `dhclient` should result in a working network connection, as shown in the following command. However, this will be lost after a reboot.

```
dhclient eth0
```

Use nano to open the `interfaces` file at `/mnt/etc/network/interfaces` and make the following addition at the bottom:

```
auto eth0
iface eth0 inet dhcp
```

Generally, a similar section exists for the loopback device.

If a static IP configuration is required, the following can be used as an example (do replace the proper values with all numbers for the desired network).

```
auto eth0
iface eth0 inet static
     address 192.168.0.10
     network 192.168.0.0
     netmask 255.255.255.0
     broadcast 192.168.0.255
     gateway 192.168.0.1
```

When using a static IP address, the system also needs to be told how to resolve things; the `resolv.conf` file at `/etc/resolv.conf` is responsible for this. Note that this file will get overwritten if the network is configured either via `dhcp` or via a network manager. Using nano as an example editor, open the `resolv.conf` file at `/mnt/etc/resolv.conf` to add the following lines to it:

```
search homedomain.local
nameserver 192.168.0.1
```

Also here, we need to properly replace the values with whatever is appropriate for the network used. Remember to save the file using *Ctrl* + *x*.

If there is more than one nameserver on the local network, a new line prefixed with the word nameserver should be used for each additional nameserver.

In the unlikely event that there is no nameserver available on the local network, Google's or OpenDNS's nameservers can be used. For Google, they are `8.8.8.8` and `8.8.4.4`, and for OpenDNS, they are `208.67.222.222` and `208.67.220.220`.

Finally, the system needs to be named on the network—the so-called `hostname`. Simply write a name that is unique on the network in the `hostname` file at `/mnt/etc/hostname`, as shown here:

```
[root@packt ~]# echo "PacktPublishing" > /mnt/etc/hostname
```

Another thing that needs to be set up is the so-called hosts file. It serves as the most basic way to look up a hostname, for example, when there is no DNS server available. The hostname needs to be in here in addition to any other hostnames that are required to be available from the network; for example, there is a time server on the network from where all the computers get their current time. Every system queries this server via `time.example.com`. Even if there's no Internet connectivity and no DNS service available, to ensure the time server is always able to be looked up, an entry to the hosts file can be added. With `192.168.0.15` being the local time server, the following command can be used as an example for the hostname and a guide to add additional hosts. Note that quite often, there is little need to add additional hosts, as DNS is nearly always used for this purpose. Remember to save the file using *Ctrl* + *x*.

```
[root@packt ~]# nano /mnt/etc/hosts
127.0.0.1       localhost
127.0.0.1       PacktPublishing
192.168.0.15    time.example.com
```

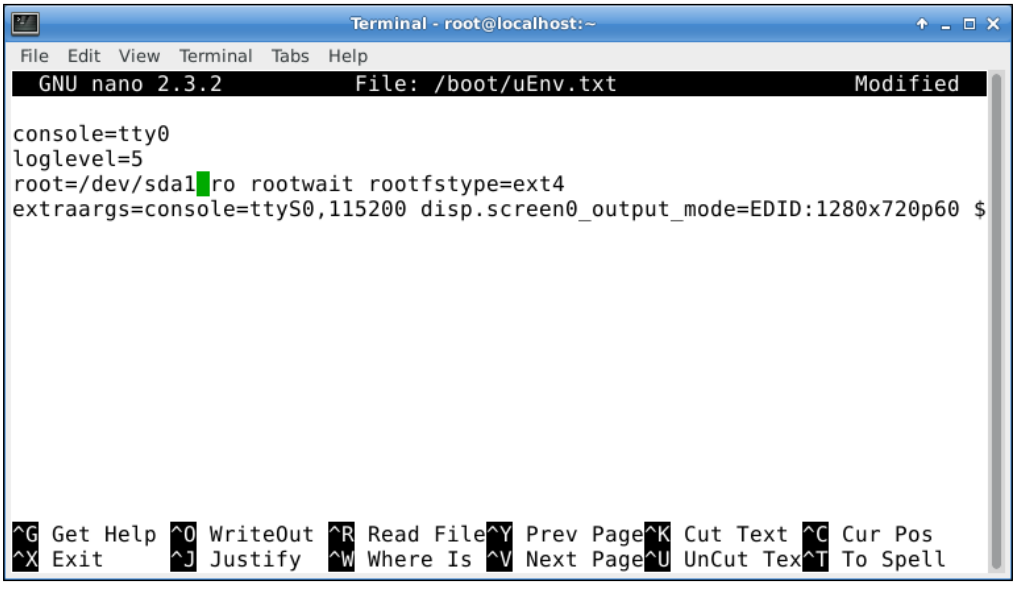
```
::1             localhost ip6-localhost ip6-loopback
fe00::0         ip6-localnet
ff00::0         ip6-mcastprefix
ff02::1         ip6-allnodes
ff02::2         ip6-allrouters
```

Making the destination medium bootable

Unfortunately, as mentioned before, the SoC cannot boot from SATA drives or USB flash drives; it requires a helper component. In this book, a small microSD card is used for this purpose. As such, it will hold the bootloader, the kernel, and a little bit of configuration to glue it all together. In the previous chapter, the installation script used for the Fedora installation was done automatically. The kernel that will be installed onto the microSD card will, by default, continue loading the rootfs from the microSD card. This will obviously need to be adjusted so that the microSD card will boot the newly created medium. To do this, the boot partition needs to be mounted first, as shown in the following command:

```
[root@packt ~]# mount /dev/mmcblk0p1 /mnt/boot
```

Using nano, edit the `uEnv.txt` file and modify the line that starts with root `/dev/sda1`, as shown in the following screenshot. Remember to save the file using *Ctrl* + *x*.



With the microSD card set up to boot from the newly created medium, it is safe to unmount it again with the following command:

```
[root@packt ~]# umount /mnt/foo
```

If things go wrong and the Cubieboard refuses to boot, the microSD card can be used in a microSD-to-USB adapter and the `uEnv.txt` file can be opened with a locally installed text editor. The OS used to make this modification, however, will need to be able to read and write `ext4` or at least `ext2` filesystems.

The root user

While a root user exists with any default Debian or Ubuntu installation, the question arises as to how to log in as the root user. The following are the two options:

- Precreate a regular user that has administrative rights via `sudo` and don't allow the root user to log in
- Or the easier way, set up a root password for the root user and use it

Security-wise, the first option is safer. Both options will be briefly covered here. It is up to the reader to decide which option is better suited and how important the security aspect of it all is. This book is also not about properly securing a system. This is left to the reader as an exercise and is far beyond the scope of this book.

Preparing the chroot command

To set up a root password, a few steps are required, as this has to be done actually from within the system. The `chroot` command makes it possible to actually enter the system as if it was booted as such. But there is a prerequisite, that is, certain dynamic directories need to be populated, namely, `/dev` and `/proc`, as shown here:

```
[root@packt ~]# mount --rbind /dev /mnt/dev
[root@packt ~]# mount none -t proc /mnt/proc
```

Here, the existing `/dev` mount is reused whereas the `proc` filesystem is mounted normally. Now, it is possible to use `chroot` into the filesystem, as shown here:

```
[root@packt ~]# PATH=/bin:/sbin:/usr/bin:/usr/sbin chroot /mnt
/bin/bash
root@packt:/#
```

Changing the root password

The `passwd` command will be used to start the password change for the root user, where a new password is entered twice. The system will not echo anything back to the user, as shown here:

```
root@packt:/# passwd
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
root@packt:/#
```

Creating a new super user

To create a new user, the `useradd` command is used. There are a lot of options to this command and it is up to you to get familiarized with them. The options used in this example are, however, the ones generally used. Besides a new user, a new group matching the user's username will also be automatically created; the `-U` flag is responsible for that. Additionally, the `-s` flag is used to supply an alternative login shell. This is completely optional but recommended as Debian defaults to the default `sh` shell, which is rather limited. The `-m` flag creates a directory for the user and copies some basic files. In the following example, the username used will be `packt`:

```
root@packt:/# useradd -m -G sudo -s /bin/bash -U packt
root@packt:/#
```

To give the new user administrative powers, it will need rights to the `sudo` command. The user has already been made a part of the `sudo` user group via the `-G` parameter. But the command needs to be actually available to be usable. While the usage of `apt` will be discussed later in this chapter, use the following command to install `sudo`:

```
root@packt:/# apt-get install sudo
```

Finally, the user will also require a password to actually log in. The following `passwd` command can be used for this:

```
root@packt:/# passwd packt
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
root@packt:/#
```

Exiting chroot

The `chroot` environment can now be exited using the `exit` command, as follows:

```
root@packt:/# exit
exit
[root@packt ~]#
```

Adding the serial console

If the system were to be rebooted at this point, the login console would show up on `tty0`, which is the normal console when a monitor and keyboard are connected. There might, however, be a case where the connected monitor is not immediately compatible or where for some reason the USB keyboard or monitor cannot be used. The serial console has served us very well until now, and thus, in order to enable it on this, the Debian or Ubuntu installation is strongly recommended. Here, the first difference between Debian and Ubuntu, however, becomes apparent. Debian still uses the older `sysvinit`, whereas Ubuntu still uses `upstart`, and while both are slowly in the progress of migrating to `systemd`, this is not applicable at this moment. Ironically, with `systemd`, or at least with the one that is installed and currently running, the Fedora image has the serial console set up by default.

Adding the serial console to Debian

The file responsible for spawning the various `init` services is `inittab` at `/etc/inittab`, and uses nano; this can be edited to add the serial console. Find the following section, uncomment the line starting with `#T0`, and remove the hashtag. Also note that by default the baud speed is `9600`, and our entire setup assumes a baud speed of `115200`, so make sure that this change has been performed, as shown in the following example. Remember to save the file using *Ctrl* + *x*.

```
[root@packt ~]# nano /mnt/etc/inittab
# Example how to put a getty on a serial line (for a terminal)
#
T0:23:respawn:/sbin/getty -L ttyS0 115200 vt100
#T1:23:respawn:/sbin/getty -L ttyS1 9600 vt100
```

Adding the serial console to Ubuntu

For `upstart`, the situation is completely different. The files in the `/etc/init/` directory are parsed by `upstart`. First, copy the file `tty1.conf` to `ttyS0.conf` as this makes editing much easier; the files are similar after all, as shown here:

```
[root@packt ~]# cd /mnt/etc/init/
[root@packt init]# cp tty1.conf ttyS0.conf
```

However, a few changes need to be made to this file. First, replace all occurrences of `tty1` with `ttyS0`. Next, in addition to `2345`, add `1` to the number of run levels. Also, remove the `and (…)` section. Finally, the `getty` line needs to be adjusted to listen to a serial port at `115200` bps. The following command shows you how the file eventually should look like. Alternatively, a new file can be created with the following content. Remember to save the file using *Ctrl* + *x*.

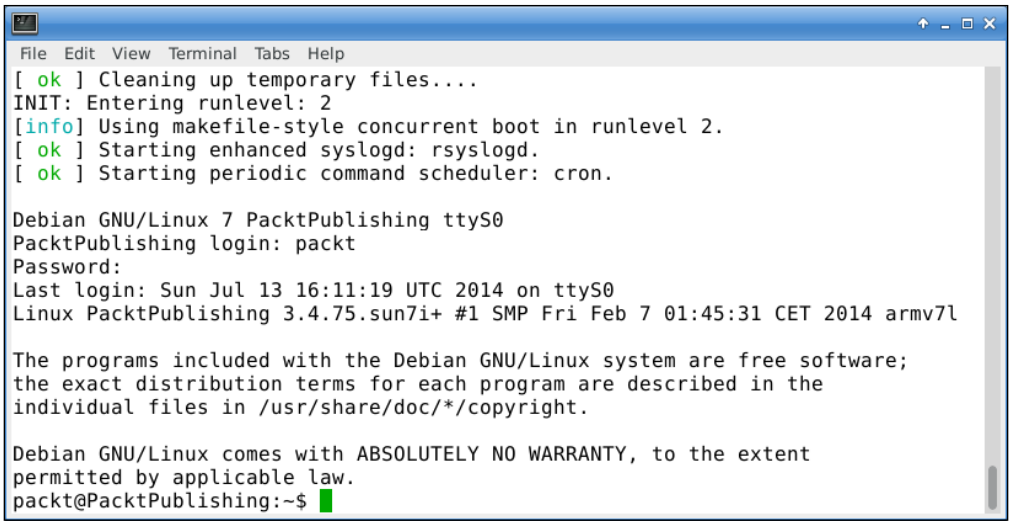
```
# ttyS0 - getty
#
# This service maintains a getty on ttyS0 from the point the system
# is started until it is shut down again.

start on stopped rc RUNLEVEL=[12345]
stop on runlevel [!12345]

respawn
exec /sbin/getty -L 115200 ttyS0 vt102
```

Rebooting the new OS

With all the required changes in place, it is time to reboot into the new operating system. First, unmount all partitions that have been mounted for this installation. Make sure that none of the mounted directories are in use. The `umount -l` command is used, where the `-l` parameter stands for lazy, which means that `umount` will try to unmount all subdirectories first and finish with unmounting the requested directory. In case of errors, manual unmounting is probably required. The `reboot` command will then reboot the system, as shown here:

```
[root@packt etc]# cd
[root@packt ~]# umount -l /mnt
[root@packt ~]# reboot
```

After the reboot, you will be greeted with a login prompt, as shown in the following screenshot. Obviously, this differs slightly between Debian and Ubuntu. After logging in with either the created user or the root, the installation part of this chapter is completed, congratulations!



Getting around the new OS via the command line

If you are new to GNU/Linux in general, using the cheatsheet as shown in *Appendix B*, *Basic Linux Commands Cheatsheet*, will be helpful, as a few common Linux commands can be explored, and it can be considered as a beginner's guide to GNU/Linux. This section will focus a little more on common tasks, keeping Debian or Ubuntu up to date and installing new software.

Introducing apt

Both Debian and Ubuntu rely heavily on apt for their software needs. Apt is a suite of commands that allows the installation of new software packages or keeping the existing ones up to date. Apt works closely together with dpkg although a regular user will probably never invoke dpkg directly. Apt is responsible for downloading a requested piece of software, checking what its dependencies are, and telling dpkg to install them. It is the Swiss army knife of package management under Debian and Ubuntu. It was the command-line AppStore before AppStores even existed.

Configuring apt

Apt does not require a whole lot of configuring. What apt does need is a list of places where it can check and download software from. The Debian and Ubuntu defaults might not offer everything of interest. The file responsible for apt's configuration is `sources.list` at `/etc/apt/sources.list`, and if you are logged in as a regular user, it requires `sudo` to grant the user additional privileges to edit the file. In the following example, the non-free component and the security repository will be added to the main component. Note that the various sources in the `sources.list` file will vary between various suites or derivatives, such as Ubuntu.

```
packt@PacktPublishing:/etc/apt$ sudo nano sources.list
[sudo] password for packt:
deb http://ftp.nl.debian.org/debian wheezy main non-free

deb http://security.debian.org/ wheezy/updates main non-free
deb-src http://security.debian.org/ wheezy/updates main non-free
```

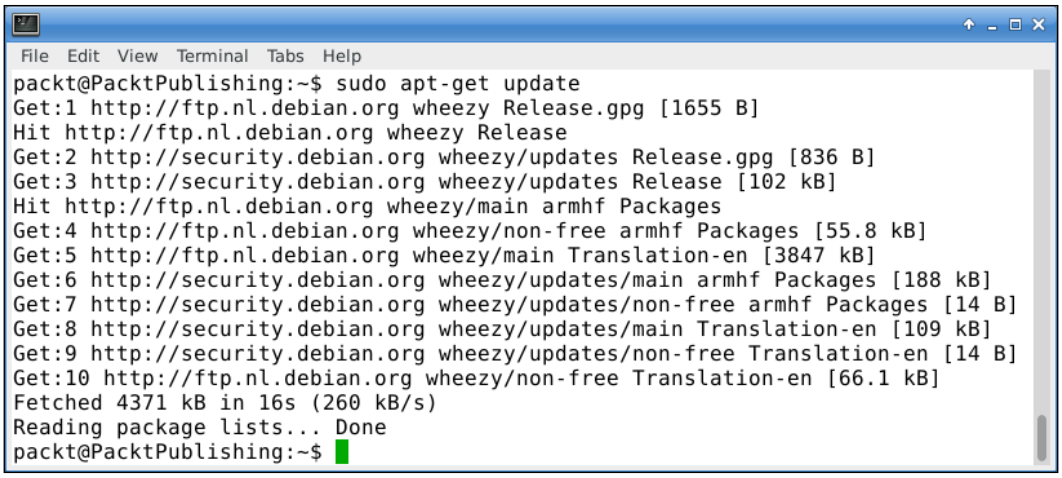
With these changes in place, apt will need to be updated, but that is handled in the next subsection.

Debian has a good tutorial for more in-depth reading at `https://wiki.debian.org/SourcesList`, and an additional components and repositories are listed at `https://wiki.debian.org/UnofficialRepositories`. For Ubuntu, the source list can be found at `https://help.ubuntu.com/community/SourcesList` and additional repositories at `https://help.ubuntu.com/community/Repositories/Ubuntu`. Further configuration beyond what is listed in this subsection is left as an exercise to the reader. Note that while in theory Ubuntu and Debian repositories can be mixed, it is not recommended and will likely cause issues.

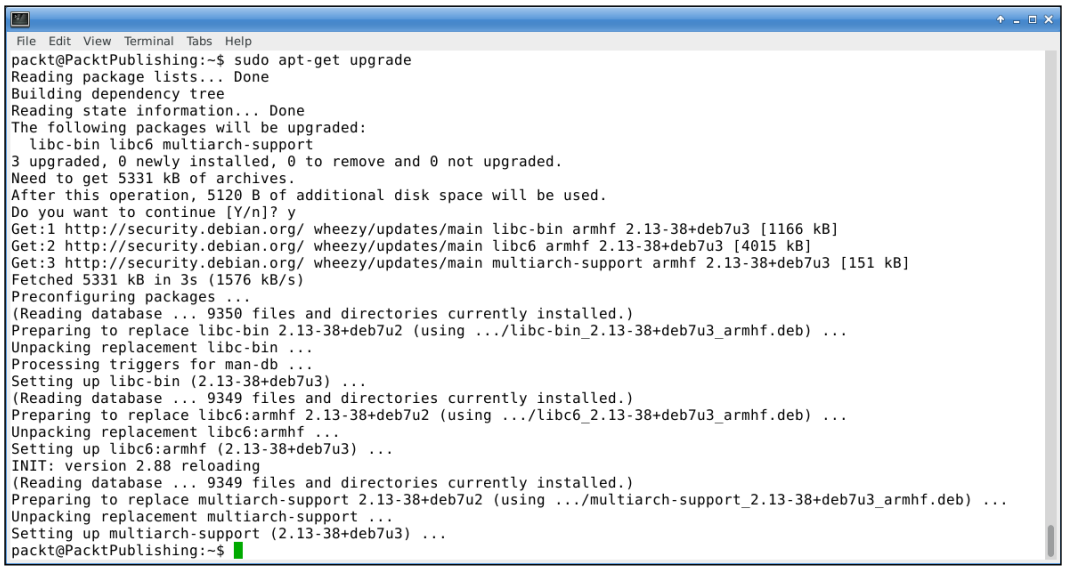
Keeping the OS up to date

Regularly checking and installing updates can be critical to security. Also, new versions of existing software packages potentially yielding new features or fixing bugs are also obtained in this way. The steps are identical for Ubuntu and Debian.

To download and update apt with a new list of the software, the first apt command `apt-get` is introduced. The `apt-get` command without parameters, however, will not do a lot more than print a help screen. As the intended action is to update the list of applications that will be the `update` parameter passed. Again, if run as a regular user, it should be prefixed with `sudo`. Also, if the network has not been configured yet and the intention is to let the graphical user interface configure the network, remember to run `dhclient eth0` to obtain a temporary network configuration. The following command in the screenshot runs an update of apt:



If sections and components were not added in the previous subsection, the list would obviously be shorter. At this point, the apt has an up-to-date list of the available software. To upgrade all installed packages, most importantly because of security updates, the `upgrade` parameter to `apt-get` is used, as shown in the following screenshot:



In the preceding example, there was only one update available, which was an update to `libgnutls`. This list can vary depending on the amount of updates, naturally.

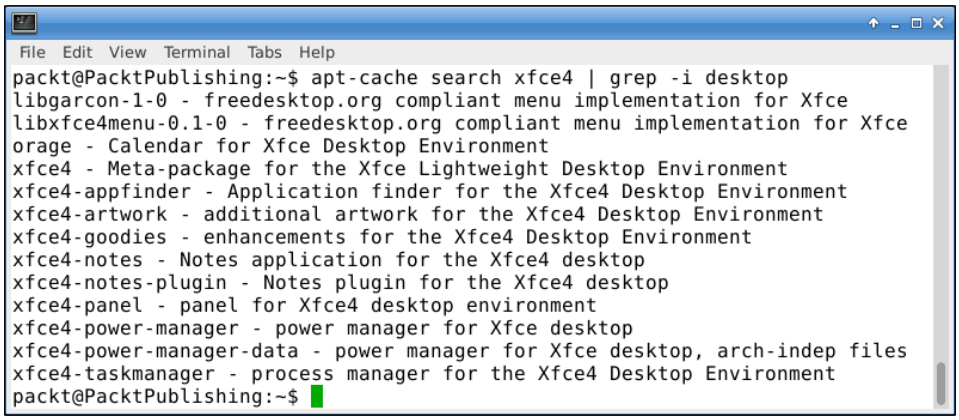
The `apt` file at `/etc/cron.daily/apt` is responsible for the daily update of apt and installing security critical packages, thus running the `apt-get` update is not always required.

Installing additional software

The most exciting thing about any OS probably is the software. However, the default created Debian or Ubuntu installation lacks most of the software. It is, after all, just the bare minimum.

Finding packages

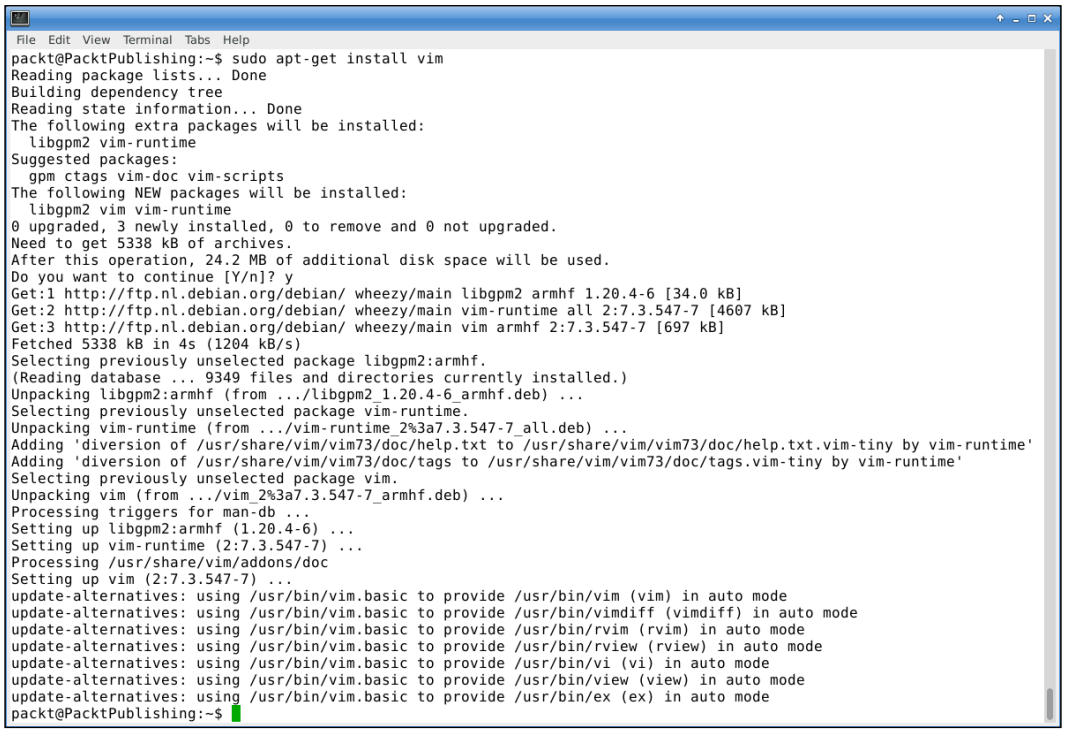
While installing packages, a problem can occur when a name is roughly remembered but not to exactly. In that case, there is an apt tool called `apt-cache`. Using the `search` parameter, the internal cached apt database can be searched for available packages. Especially in combination with the `grep` command, these two can be helpful to find what is needed, as shown in the following screenshot. The `grep` – command is explained in *Appendix B*, *Basic Linux Commands Cheatsheet*.



Additionally, you might want to search for a filename of an application. There is an apt tool for that as well. The `apt-file` tool using the `search` parameter will allow you to search for files inside packages. Unfortunately, at the time of writing this book, the `apt-file` tool is not yet included in the current stable release of Debian or Ubuntu, but should be hopefully added soon.

Installing the software package using apt-get

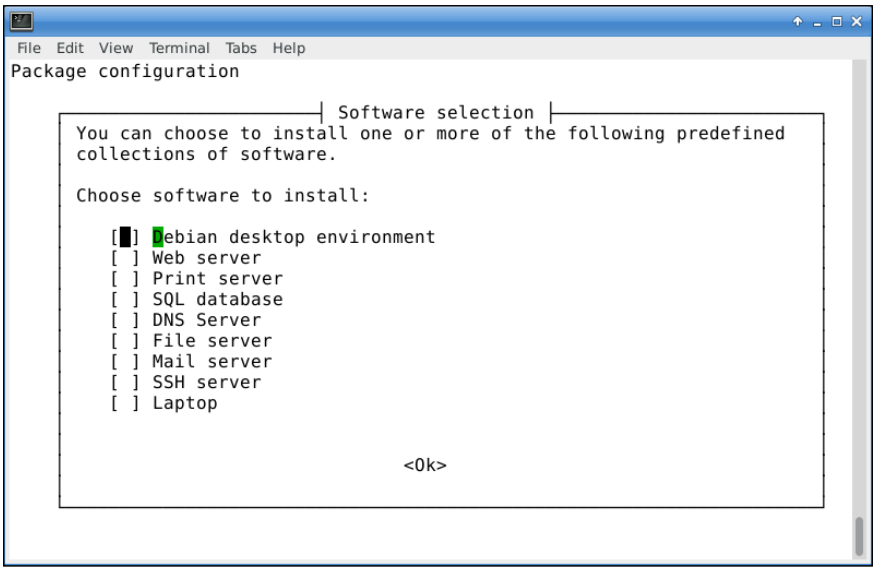
The most basic way to install a software package is also via `apt-get`, the parameter not surprisingly being install. Nano has been used often as an example editor. This is because it is very easy to use and nearly always preinstalled as it is so small. **Vi** is another small editor that is nearly always preinstalled but is far from easy to use. Vi has a bigger brother called Vi improved or vim. Let us install vim via `apt-get`, as follows:



After downloading and installing a few packages, vim is now installed.

Installing the software package using tasksel

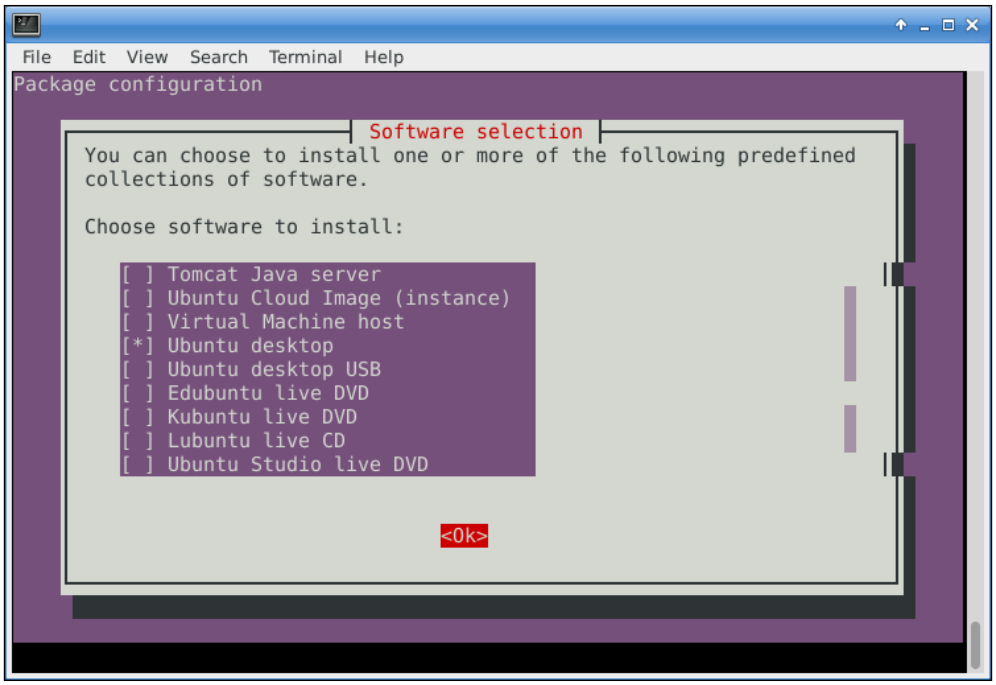
More than often, a collection of packages is required to have the system perform certain functions. The `tasksel` command can be used to install a collection of packages to perform a certain task. Running `tasksel` with elevated privileges yields the menu, as shown in the following screenshot. Ubuntu does not have `tasksel` preinstalled and requires it to be installed via `apt-get`, as shown in the previous example for vim.



Installing the Debian desktop environment task will install a graphical desktop environment based on **GNOME** and some additional packages marked as standard by Debian, such as LibreOffice.

The downloading and installation might take a while depending on the target medium and Internet connection speed. Installation on an SSD with a very fast Internet connection can take about thirty minutes.

Under Ubuntu, `tasksel` will look slightly different, but even here there is an Ubuntu desktop option, as shown in the following screenshot. This will take about the same time thirty minutes or more.



Installing packages via metapackages

The tasks available via `tasksel` appear to be rather limited. Instead, it is probably easier to use metapackages. Metapackages are, in effect, not that much different from tasks; in fact, they very well might be the same in the background. A metapackage is not really a package that installs anything, rather it is a list of packages or a collection of packages that get installed from it. For example, `xfce4` is a metapackage for Debian, which will install all the packages required for the `xfce4` desktop environment and will also install software that the package maintainers thought would make sense to have for a full-fledged desktop environment, such as a file manager.

For Ubuntu, this metapackage is just called xfce, though the xubuntu-desktop metapackage should be more interesting here.

Another interesting metapackage to match the `xfce4` metapackage is `xfce4-goodies`. In fact, the following command will show you how to install multiple packages in one go. Running the command will result in a huge list of packages to be installed, but will yield a usable `xfcef4` desktop, as if it were installed from a CD, for example. One could even argue that an installation CD would do just that, as shown here:

```
packt@PacktPublishing:~$ sudo apt-get install xfce4 xfce4-goodies
```

After waiting about 30 minutes for the xfce4 desktop to get installed, xfce4 can be started using the `startx` command. At this point, however, the monitor and keyboard will have to be used. Xfce4 cannot be used or started over the serial console. This yields an almost usable desktop environment. Almost means here that certain things with permissions are missing. A user is not allowed to shut down the machine by default, for example.

There are several ways to allow these things to work properly, and one is the use of a login manager; xfce4, however, does not come with one, but that is okay. There are plenty of login managers to choose from. GNOME comes with `gdm`, but going with a lightweight login manager that matches Xfce4 as a lightweight desktop manager, LightDM should be a good candidate. Installing the `lightdm` package should require no extra instructions. Rebooting the Cubieboard now will yield an active login window, which upon login, wraps up this chapter.

Summary

Having worked through this chapter, admittedly a big one, the result should technically be the same as the previous chapter; a working desktop environment based on either Debian or Ubuntu. The installation of additional software via the command line is now a breeze, and keeping the system up to date is not an issue at all.

The next chapter will take this installation and turn it into a server for various tasks, optionally retaining the desktop functionality.

5
Setting Up a Home Server

Since you are now familiar with creating an entire root filesystem from scratch, it is a good time to create something a little more complex. Granted that a desktop can be quite useful by itself, but very often, a desktop OS on these kinds of ARM development boards might be a little too slow. However, these devices are very useful for low-power home servers.

The term home server is used here, as most services are useful to a person at home, but using a Cubieboard in a colocation center and getting it to function as a web server and mail server, for example, also works quite well. However, when doing that, security does come to mind. Security is not strongly addressed in this chapter, as it is a subject that requires a lot of attention and strong knowledge of security, both of which are out of the scope of this book. It has to be said that these instructions are not specific to a Cubieboard at all, and there are obviously many more services that could be thought of.

In this chapter, we will cover the following topics:

- Accessing the Cubieboard remotely
- Learning how to start, stop, and restart services
- Adding and removing a service to be started at boot
- Automatically running tasks at scheduled times
- Set up various services (Squid, Apache, Samba, transmission, and ownCloud)

Prerequisites for the home server board

In this chapter, several software packages will be installed upon the previous Debian or Ubuntu installation. If this installation has seen too much wear and tear due to experimentation, it is recommended to go over *Chapter 4*, *Manually Installing an Alternative Operating System*, again.

It is wise to skip the final segment, where a graphical desktop environment is installed, as it will serve no purpose in this chapter. The initial Fedora installation on the SD card can also be reused, but it is up to you to identify the differences between these distributions.

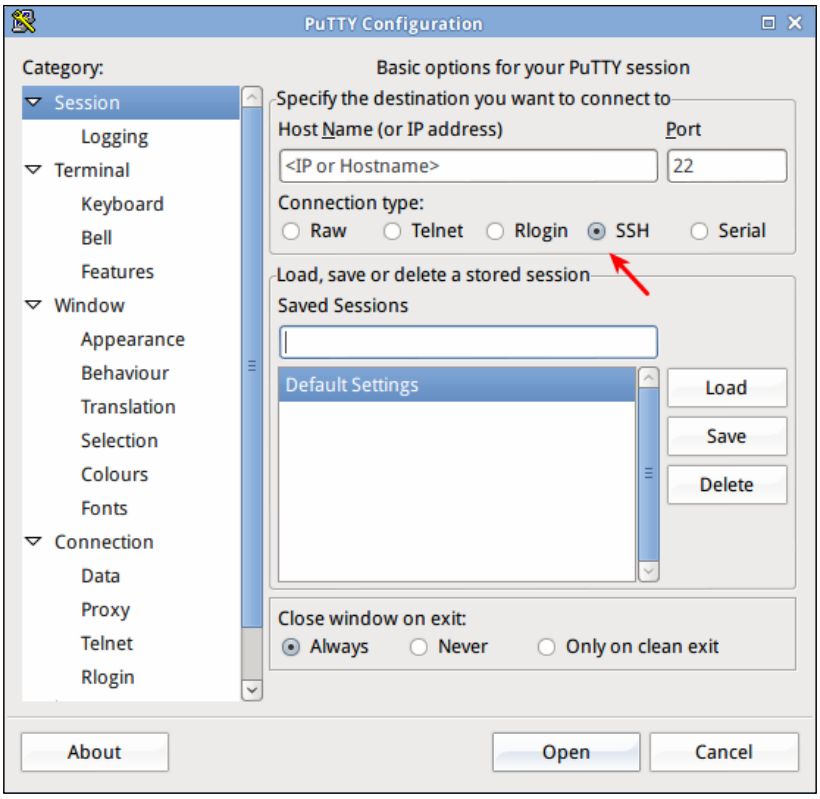
Accessing the server remotely

Most of the time, if not all the time, the Cubieboard has been accessed via a serial console, or directly via the keyboard and mouse while it is connected to a monitor. This works fine, but after the setting up and tinkering is done, it might be interesting to get it to sit in a different room where it can perform its task unattended. It could even be possible to have several boards running in a dataserver, for example, where remote access might become crucial. Thus, having remote access is very useful.

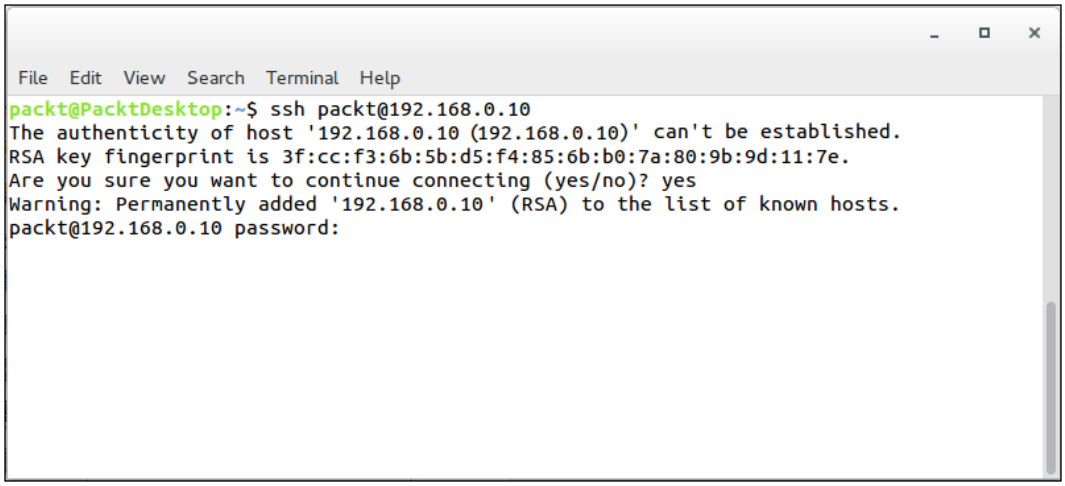
The most common and well-established way to connect to a Debian or Ubuntu machine is via an `ssh` server. Installing an `ssh` server is simple; just use the following command:

```
packt@PacktPublishing:~$ sudo apt-get install openssh-server
```

Just as with UART access, PuTTY can be used for `ssh` access. However, the important thing here is to set the connection type to `ssh`. The port should be set to `22` by default, and the correct IP or hostname needs to be used. See the following screenshot for an example of this:



If you are using an existing Linux command line, `ssh` can be invoked quite easily, as shown in the following screenshot:



The first time you connect to a host, the `ssh` fingerprint is displayed followed by a question as to whether you are sure you want to connect. In this case, it is safe to accept the proposal to continue.

When connecting over the Internet, it is recommended that you always verify the key. If someone messed around with the server or tried to perform a man-in-the-middle attack, this key will no longer match, and you will know that something is wrong.

After entering the password, the familiar `packt@PacktPublishing:~$` command prompt will appear.

Interacting with services

On a server, it is quite common to start, stop, or restart a service. This is not only while trying to fix a problem, but also sometimes to reload a configuration, for example. It might be that a service is used to run only occasionally but not every time during booting. For these purposes, the so-called startup scripts are available. When a service is installed, usually, a script to control its behavior is placed at `/etc/init.d`.

For a server, it usually makes sense to configure networking either statically or via a DHCP server directly in the `interfaces` file. This to ensure networking is always available, as a graphical desktop is often not installed.

Let us take the networking script, for example. It is responsible for bringing up a network device, but only those configured at `/etc/networking/interfaces`. Refer to *Chapter 4*, *Manually Installing an Alternative Operating System*, on how to properly set up a network configuration; it is assumed that `eth0` is configured in this way.

Stopping the networking service will deactivate all the networking interfaces! While this does not have to be a problem, be careful when using this command if you're using the connection remotely.

Starting, stopping, restarting, or reloading a service

To stop the network interface, issue the following command. If you are logged in as root, which is not recommended, omit the `sudo` command.

```
packt@PacktPublishing:~$ sudo /etc/init.d/networking stop
Deconfiguring network interfaces...done.
```

Similarly, the network interface can be started again using the `start` parameter, as follows:

```
packt@PacktPublishing:~$ sudo /etc/init.d/networking start
Configuring network interfaces...
```

The same can be done using the `restart` or `reload` parameter, where `reload` forces the service to reload its configuration.

Some distributions, such as Ubuntu, might use upstart or even SystemD. Sometimes, scripts are left behind in the `/etc/init.d` directory that can be used as described previously, but this is not always the case, and thus it might be required to get familiar with SystemD or upstart.

Adding or removing a service from the boot up

Sometimes, it is required to always make a service start during boot time. For example, in the previous chapter, `lightdm` was installed to provide a graphical login service. This service was automatically added to be started during booting. Preventing `lightdm` from starting during booting any longer can be accomplished using the `update-rc.d` command. Passing the optional `-f` flag forces removal, as shown here:

```
packt@PacktPublishing:~$ sudo update-rc.d -f lightdm remove
update-rc.d: using dependency based boot sequencing
```

The graphical login manager will now no longer be started after a reboot using the preceding section to stop `lightdm` immediately; a reboot is then not even needed.

Adding a service is quite similar; instead of asking it to be removed from the boot time services, the default keyword asks that the service be started during booting, as shown here:

```
packt@PacktPublishing:~$ sudo update-rc.d lightdm defaults
update-rc.d: using dependency based boot sequencing
```

Defaults indicate that all the default runlevel services should be started or stopped. A Linux system has various runlevels, and you are encouraged to read more about it.

After each reboot, `lightdm` will be started. The preceding section was all specific to Debian. Ubuntu, however, has backwards-compatible scripts for most services in `/etc/init.d` and can be used as described previously.

As mentioned in the earlier chapter, both Debian's SystemV and Ubuntu's upstart will, in time, be replaced by SystemD or both. It is up to you to learn more about SystemD when the time comes.

Running scheduled tasks automatically

It is often necessary to have certain tasks run at scheduled intervals. Think of downloading and updating a virus database, for example. For this purpose, Linux systems are equipped with a program called **Cron**. Cron normally runs in the background and checks whether it was given any jobs to run at a certain time. While it is up to you to get to know Cron in detail, by default on Debian, Cron makes running Cron-jobs very easy by supplying four directories in `/etc`, namely, `cron.hourly`, `cron.daily`, `cron.weekly`, and `cron.monthly`. From the names, it should be obvious what their intentions are. Placing an executable file here makes Cron run the command hourly, daily, weekly, or monthly.

A simple example is to ask `apt-get` to update its local database every day; a simple script can be created for this purpose. Using the editor as root, create a new file called `apt-update` in `/etc/cron.daily/`, as follows:

```
packt@PacktPublishing:~$ sudo nano /etc/cron.daily/apt-get
#!/bin/sh
apt-get update
```

The first line is important, as it tells the system that this is a shell script, or rather, that `/bin/sh` needs to be used to execute the lines in the rest of the file. Cron runs all its jobs in `/etc/cron.*` by root by default, so there is no need to prepend it with `sudo`.

With the file saved, it is important to give it the permission to execute. By default, it is just a text file. Cron will only execute files with the proper permissions. This is easily fixed using `chmod` to mark this newly created script executable, as shown in the following command:

```
packt@PacktPublishing:~$ sudo chmod u+x /etc/cron.daily/apt-update
```

It needs to be said, however, that while this was a very useful example, Debian and Ubuntu by default update the apt repository on a daily basis. So, while this serves well as an example, it is recommended that you clean it up afterwards, as shown in the following command:

```
packt@PacktPublishing:~$ sudo rm /etc/cron.daily/apt-update
```

Setting up a proxy server

When bandwidth is scarce and has to be shared with multiple members of a household, and on top of that, a reduction of ads is desired, a proxy server can offer a solution. Additionally, it allows Internet access without complicated firewall rules; just configure the browser to use the proxy. In the following subsections, the steps to set up a proxy server are explained.

Installing Squid

The proxy server used in this book is called Squid, and as you learned from the previous chapter, it can be installed easily with the following command:

```
packt@PacktPublishing:~$ sudo apt-get install squid
```

Squid in Debian comes with a reasonable default configuration file at `/etc/squid/squid.conf`. Using nano or any other editor, the file can be opened and examined. The squid configuration file is very heavy on comments, working as a guide or manual.

By default, Squid will only listen to the traffic on connections that it thinks are from the internal network. The lines starting with `acl localnet` followed by the internal network, IP-range, indicates this. It should be noted that an additional `localnet` line can be added, but be careful—adding an IP-range internal network that is not local can mean that the proxy is accessible worldwide, so do be careful when experimenting here. This, however, only defines the `localnet`; it has not been granted access yet. Searching through `squid.conf`, a commented section reading the following command will be found:

```
#http_access allow localnet
```

This line allows HTTP access where the source originates from `localnet`, which we just learned about, and which was defined as the internal network range. You can uncomment it by removing the hashtag, or the number sign; this grants access to the so-called `localnet` line, and after restarting Squid, the proxy should now allow usage from the internal network.

Setting up a caching proxy

A proxy server can temporarily store frequently accessed data so that it only needs to be downloaded over an Internet connection once. This can, in certain cases, greatly reduce bandwidth usage; for example, imagine that a household of five computers with all of them requiring to download a certain update file. If all the five computers download this update via the proxy, the proxy only downloads this the first time the file is requested. The moment a second computer requests that same file, the proxy would offer its internally stored file making the download much faster and not burden the Internet connection.

By default, Squid uses a cache of 100 MB only. Take a look at the `cache_dir` parameter to learn more about this.

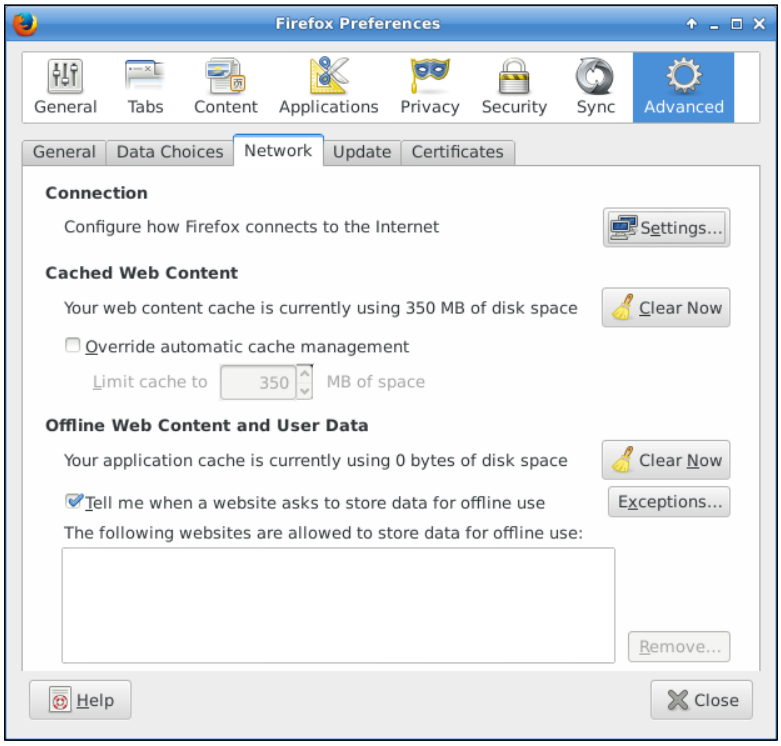
Squid does not require any additional configuration to function as a caching proxy. After it has been installed, it is automatically started and configured to be started upon a reboot. To change any of these behaviors, refer to the earlier two subsections.

Configuring a browser to use the proxy

Without covering every browser or every operating system out there, at least one example will be shown here. In this case, we choose Mozilla Firefox. To configure the browser to use the proxy, follow these steps:

1. First, open the **Preferences** option, which, depending on the operating system used, is under **Edit** or under **Tools**.

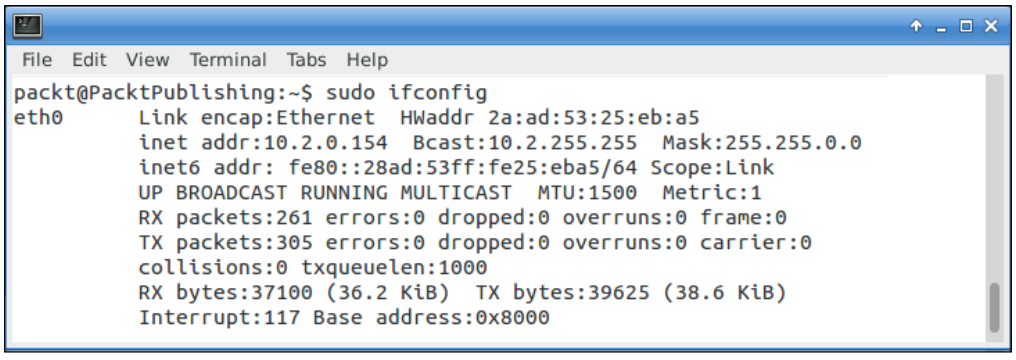
2. In the preference screen, navigate to the **Advanced** tab, and from there, to the **Network** tab. The first button on the right reads **Settings**, and it holds the network configuration, as shown in the following screenshot:



3. In the **Settings** dialog, select **Manual proxy configuration**, which will enable the edit boxes.
4. For the **HTTP Proxy**, the hostname or the IP address needs to be entered. The default port for Squid is **3128**.

5. Finally, the **Use this proxy server for all protocols** option can be safely enabled, as Squid covers them all, as shown in the following screenshot:

6. If the IP address is unknown, it can be obtained using the `ifconfig` command on the Cubieboard where Squid is running. In this case, the IP address is `192.168.0.10`, as shown here:



It is not strictly required to use `sudo` to access the `ifconfig` information. Most of the time, it can be accessed via its full path, `/sbin/ifconfig` in this case, without requiring elevated privileges. Additionally, `ifconfig` might be replaced by the `ip addr` command on certain distributions.

Starting a web-surfing session should now go via the proxy. Manual configuration of a proxy is not strictly required. As seen here, there is the **Use system proxy settings** or **Auto-detect proxy settings for this network** option. When using the first option, whatever is configured by and for the operating system is used as the default proxy by the browser. The latter option is used if the proxy settings can be detected via the so-called WPAD, which is up to the reader to learn more about and can be found at `http://en.wikipedia.org/wiki/Web_Proxy_Autodiscovery_Protocol`. Both of these options, however, are not covered here.

Setting up a blocking proxy

One of the advantages of using a proxy is that certain sites can be easily blocked. Let us say the sites `hotmail.com`, `ebay.com`, and `live.com` need to be blocked. This list of domain names will have to be stored somewhere, for example, `blocked.domains.acl` in the `/etc/squid` directory. To do this, follow these steps:

1. Start editing the file and add those names, as follows:

   ```
   packt@PacktPublishing:~$ sudo nano
   /etc/squid/blocked.domains.acl

   hotmail.com

   ebay.com

   live.com
   ```

2. Next, go to a very specific section in the configuration file, as the order does matter. Find the section called `# TAG: acl`, and scroll beyond the big commented section where the `acl` tag gets documented. The `acl localnet` definitions will show up as seen in the earlier *Installing Squid* section. Just before the next section starts, find `# TAG: http_access` and add the following line:

   ```
   acl blockeddomain dstdomain "/etc/squid/blocked.domains.acl"
   ```

   In short, under the section called `acl`, a new item will be added, just before the `http_access` section. Here, Squid is instructed to create an `acl` called *blocked domain* and mark the domains as destination domains from the file just created. So now there should be a list of `acls` with the blocked domain one being last.

3. Immediately following the `acl` section, the `http_access` section starts. Here, the order is crucial. Just above the sections where the localhost and `localnet` were allowed, the following command needs to be added:

   ```
   http_access deny blockeddomain
   ```

   The reason is that, first, things get denied, such as the blocked domains, and then localhost and `localnet` are granted access to what is left. Finally, everything else gets blocked for those that have not been allowed anything.

4. Using the `restart` command you learned earlier in this chapter, Squid can be restarted and will now refuse to load all the content from the mentioned sites.

Taking this example a step further, it would be nice to block all domains that do nothing but serve ads. While creating a file to list all these domains is of course a valid solution, it becomes quite tedious to list all domains and all of its permutations. Squid can instead do regular expressions on a list, making things a lot easier. To do this, follow these steps:

1. As before, create a file named `squid.adservers`, and enter the following regular expression. It's okay if you don't understand what any of it means; it's only an example for now.

   ```
   (^|\.)wikia-ads\.wikia\.com$
   ```

2. After saving the file, open `squid.conf`, and just like before, add the following lines to the appropriate sections:

   ```
   acl ads dstdom_regex -i "/etc/squid/squid.adservers"
   http_access deny ads
   ```

3. Squid is instructed to create a list following the file, but this time using a regular expression for the destination domain list. Again, Squid will then deny all HTTP access to the domains in the list ads, which in this case is `wikia-ads.wikia.com`.

Manually, adding regular expressions for each and every ad-network can be tiresome, especially when trying to maintain them. Luckily, a group of people maintains a list of known ad-servers that can be used without too much work; the list can be found at `http://pgl.yoyo.org/adservers/`. Someone was even kind enough to provide a script that can be used to download the list and prepare it to be used with Squid. Download the following file using `wget`, as seen in the following example:

```
packt@PacktPublishing:~$ wget
http://pgl.yoyo.org/adservers/scripts/squid/update-squid-
adservers.txt
```

Before being able to use this script, it does require minor modification. To do this, follow these steps:

1. Open the file, find `listurl`, and replace it as shown here:

   ```
   listurl='http://pgl.yoyo.org/adservers/serverlist.
   php?hostformat=squid-dstdom-regex&mimetype=plaintext'
   ```

2. Also, double-check the target file parameter, for it will have to match what was used on the ad's `acl` rule that was defined as shown here:

   ```
   targetfile='/etc/squid/squid.adservers'
   ```

3. After saving the file, it can be run as root, and it will download an up-to-date list of ad-servers and reload this data into Squid. Make use of `sh` here, as `sh` is run as root and it is told to run the script as if it were a shell script:

   ```
   packt@PacktPublishing:~$ sudo sh update-squid-adservers.txt
   Reloading Squid configuration files.
   done.
   packt@PacktPublishing:~$
   ```

4. To double-check whether all this worked accordingly, open the target file, probably `/etc/squid.adservers` by default, unless adjusted as suggested earlier.
5. One final step is to have this script update on a monthly basis. Using the earlier section on how to set up a Cron-job should make this task easy.

Using the `cp` command, the file can be copied to the appropriate directory. It is probably a wise idea to drop the `.txt` extension when copying.

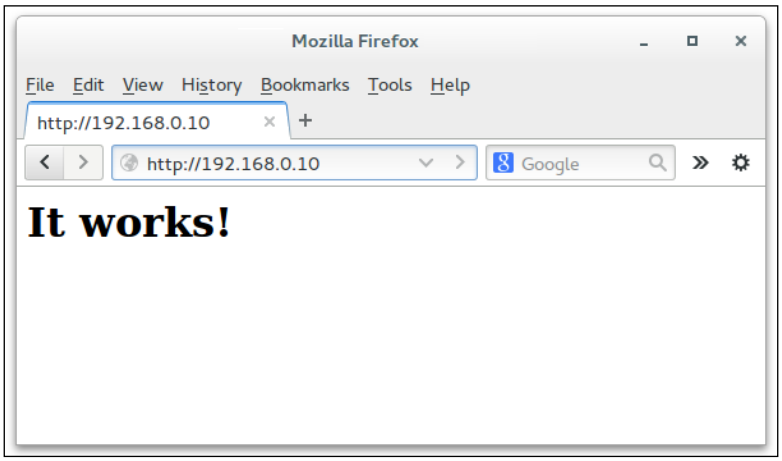
Setting up a web server

A personal web server is something that a lot of people involved in Linux have. Setting up one properly is, however, a topic worth of an entire book. In this section, only the most basic of steps will be covered for you to get started. By no means is it complete or secure, as we will rely on the default settings that Debian or Ubuntu ship with in their packages.

There are several well-known web servers, but without a doubt, the most well-known is the following Apache web server:

```
packt@PacktPublishing:~$ sudo apt-get install apache2
```

As before, Debian will automatically start Apache, and it can be used right away. Using a web browser by typing the IP or hostname, such as `http://192.168.0.10` which was used earlier, into the URL bar will open the following window:



The file being served is from `/var/www/index.html`. Feel free to edit this document or even entirely replace it.

It might be useful to give the web developer access rights to this directory and its files. Quite often, there is a group associated with the `webroot` directory. Admins often even create several groups for the various webroots they might have. For simplicity's sake, the user and group is set to our example user, as follows:

```
sudo chown -R packt:packt /var/www
```

Setting up a file server

Editing the files directly on the Cubieboard can be tiresome for sure. Even when using the previously installed Xfce desktop, once it is placed at a remote location, accessing the files remotely will be quite useful. It might be the case that the media files are desired to be shared throughout the household.

There are two approaches to this situation. With the `ssh` server installed earlier in this chapter, any modern Linux desktop environment has the ability to access files via `ssh`. While this works well, it is left to the reader as an exercise. What this section will focus on is installing and setting up Samba as a file server. To do this, follow these steps:

1. The files shared via Samba are accessible via many operating systems and in many devices it can be easily installed, as shown here:

   ```
   packt@PacktPublishing:~$ sudo apt-get install samba
   ```

2. After Samba has been installed, it is probably a good idea to create a directory that will be shared using the `mkdir` command. It can be called `mediafiles` and lives in the root of the filesystem, as shown here:

   ```
   packt@PacktPublishing:~$ mkdir /mediafiles
   ```

3. Using an editor, open the Samba configuration file at `/etc/samba/smb.conf`, as shown in the following command:

   ```
   packt@PacktPublishing:~$ sudo nano /etc/samba/smb.conf
   ```

4. One variable that probably should be changed is the so-called workgroup; this is the name of the network that Samba will try to join:

   ```
   # Change this to the workgroup/NT-domain name your Samba server
   # will part of
      workgroup = PacktNet
   ```

5. The other thing that needs to be added at the bottom of the file is a section where the previously created directory is being shared, as shown here:

   ```
   [mediafiles]
      comment = Shared media files
      read only = no
      path = /mediafiles
      guest ok = yes
   ```

Security-wise, this is not the best protection, as everybody is allowed to read and write to the media files and no valid account is required to access these files. There is one final barrier, however, and that is the permissions left on the directory. Samba will run on the media files as the user tries to access this directory.

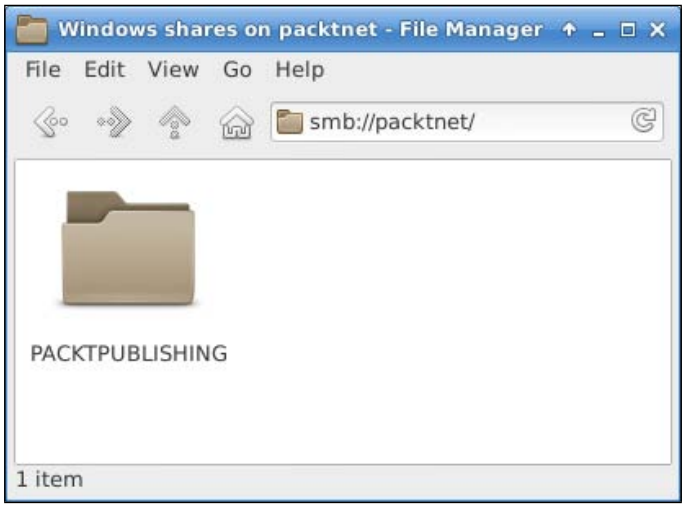
Since this share is being used as a guest user, or no user, the permissions will be applied when reading the directory media files.

1. Using `chmod` to change the permission to allow everybody to read and write makes it possible that this share can be used by everybody, as shown in the following command:

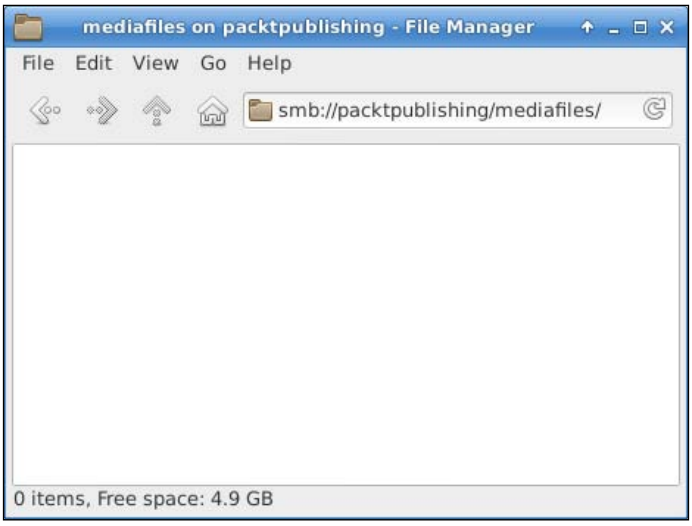
   ```
   packt@PacktPublishing:~$ sudo chmod o+rw /mediafiles/
   ```

   Within a home network, this is not a problem. Protecting things better, however, is a good exercise left to the reader. Samba is a great suite offering a whole lot more than just file sharing. Learn more at `http://www.samba.org/`.

2. After saving these changes, Samba needs to be restarted.
3. Using a file browser on a desktop, there should be a network workgroup called PacktNet, as was defined for the workgroup, and in that network, amongst other possible hosts, there should be `PacktPublishing`, the name of the host, as shown in the following screenshot:



4. Finally, within `PacktPublishing`, the freshly shared directory `mediafiles` will become visible.



Setting up a torrent server

Torrents are a common way of sharing data. Many Linux distribution ISO files are shared via torrents. Having a dedicated server dealing with torrents without having the main PC turned on is also very useful. It has to be noted, however, that some storage for that torrent data is required. Having all data on a SD card is probably not wise. Have a look at the following command to install transmission, a torrent server:

```
packt@PacktPublishing:~$ sudo apt-get install transmission-daemon
```

After the transmission is installed, it will need to be configured if the remote administration is desired. By default, the remove administration feature will only listen on the localhost. Using `sudo`, the configuration file should be opened, and two items, `rpc-password` and `rpc-whitelist`, need to be changed. The `rpc-whitelist` item determines which hosts can access the server, and the `rpc-password` item determines which password was used by any clients connecting to the server, as shown here:

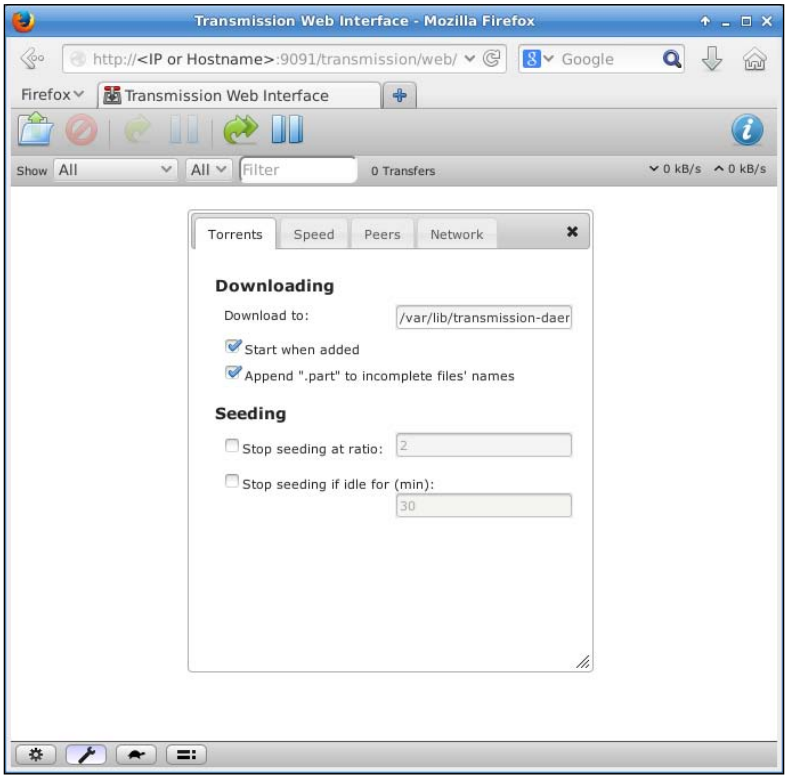
```
packt@PacktPublishing:~$ sudo nano /var/lib/transmission-daemon/info/
settings.json
    "rpc-password": "mysecretpassword",
    "rpc-whitelist": "127.0.0.1,192.168.*.*",
```

The default password set is an encrypted hash, but when modifying it, a regular plain-text password can be used. This seems like a security problem, and it is, but only very briefly. Upon being told to reload its configuration file, the transmission will read the password, encrypt it, and rewrite the configuration file.

Transmission needs to be reloaded because if the transmission is restarted, it will not re-read the configuration file but will rewrite configuration file on exit. The rest of the defaults should be satisfactory though it might be wise to double-check that `rpc-authentication-required` is set to `true` and that `rpc-enabled` is also `true`. Additionally, the `rpc-username` variable can be changed to something memorable, but do remember to use the new username in the remainder of this chapter, as shown here:

```
packt@PacktPublishing:~$ sudo /etc/init.d/transmission-daemon reload
Reloading bittorrent daemon: transmission-daemon.
```

Once the transmission is reloaded, a web browser can be used to browse to the IP address or the hostname on port `9091` and can be further configured from there, as shown in the following screenshot:



The web interface is really nice and very usable, and there are various transmission clients that can remotely connect. They work as if the client ran locally but talk to the transmission-daemon running on the network. One such program is called `transmission-remote-gtk`. For Android, for example, there exists an application called **Transdroid**.

Setting up a personal cloud

The cloud is where everything is these days. However, a lot of people are not happy with storing everything in the cloud on a random server. Luckily, there is something called ownCloud. This is a web service that lets you have your own personal cloud.

For this book, ownCloud will be installed using the SQLite database. For very light, single user workloads, SQLite will be fine.

A more serious, multiuser installation would strongly benefit from using PostgreSQL or even MySQL; however, those systems are far more complex to set up and work with. After getting more comfortable with Linux in general, this can be an exciting exercise to the reader.

Installing **Mail Transport Agent** (**MTA**) is suggested by ownCloud; However, installing MTA is also beyond the scope here. To bypass the installation of any MTA, a fake package called `lsb-invalid-mta` exists, which can be installed as shown here:

```
packt@PacktPublishing:~$ sudo apt-get install lsb-invalid-mta
```

Unfortunately, as of the time of writing this book, the ownCloud package is not yet available for Debian Wheezy, but it is for later versions. Also, Ubuntu has had it available in its repositories for a while now. In some cases, such as the ownCloud package, it is available in a different repository—the backports repository. This was covered already in the previous chapter, where the apt repository was mentioned. Adding the backports repository can be done by adding the following line. Again, use an appropriate mirror if you can. In the following example, the `nl` mirror is used:

```
deb http://ftp.nl.debian.org/debian wheezy-backports main
```

Remember to update the apt database after adding the backports repository. There are a few dependencies of the SQLite variant of ownCloud that were put in a metapackage called `owncloud-sqlite`, as follows:

```
packt@PacktPublishing:~$ sudo apt-get install owncloud-sqlite
```

With the ownCloud SQLite dependencies installed, it is time to install ownCloud itself. Unfortunately, the ownCloud package, by default, wants to install a few suggested packages for a database. In the case of Debian Wheezy, it will always want to try to install the MySQL database even though a different database will be used.

To remedy this, we introduce a new options to `apt-get`, `–no-install-suggests`, and `–no-install-recommends`, which, as the names suggest, do not install any suggested or recommended packages. But even that will not prevent the full installation of MySQL. By adding `mysql-server-` (notice the dash at the end of the list of packages to be installed), apt will not install MySQL Server, as shown here:

```
packt@PacktPublishing:~$ sudo apt-get install –no-install-suggests –
no-install-recommends owncloud mysql-server-
```

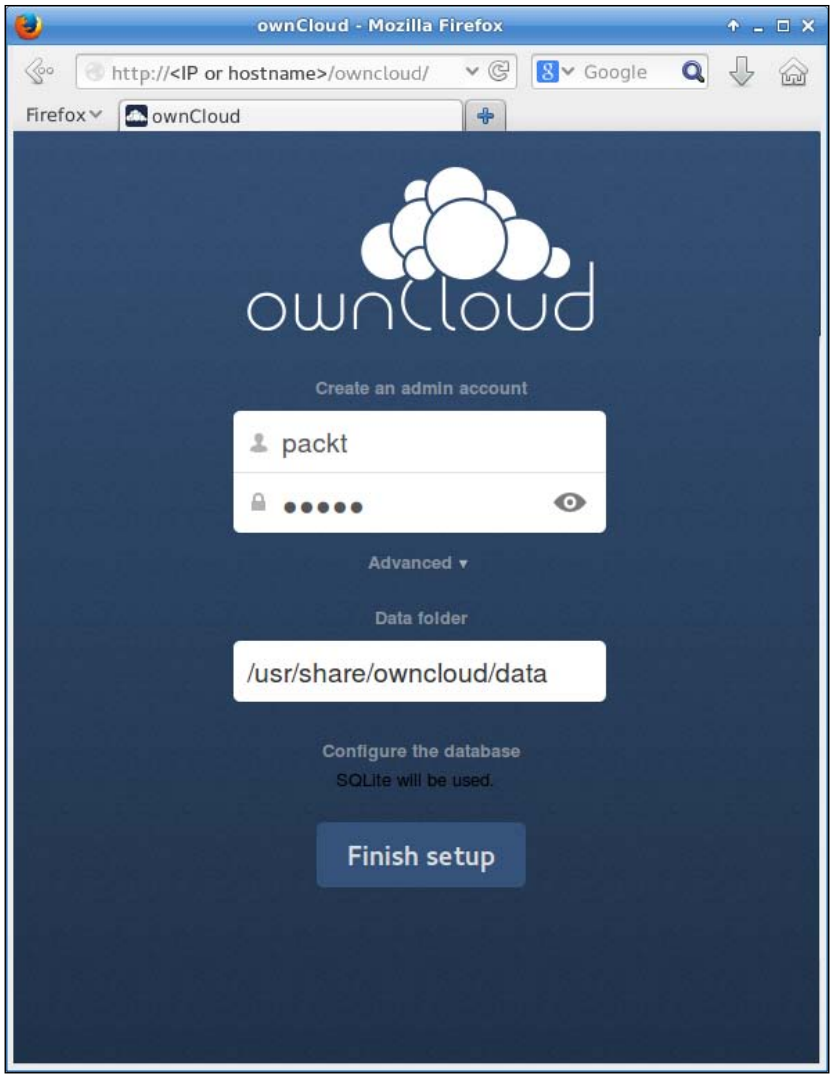
At the time of writing this book, the ownCloud package in the `wheezy-backports` repository is broken. At least a certain version of `php-getid3`, which is not available in the standard repositories, is required by ownCloud, as shown here:

```
owncloud : Depends: php-getid3 (>= 1.9.5~) but 1.9.3-1+deb7u1 is to be
installed
```

This package, however, is also available in the `wheezy-backports` repository, but apt needs to be instructed to install it specifically from there. The `-t` parameter tells apt to install a package from a specific repository, as shown here:

```
root@PacktPublishing:~$ sudo apt-get -t wheezy-backports install php-
getid3
```

It might be required to install this or other packages from the `backports` repository as a dependency of ownCloud. Using a web browser, navigate to the IP or hostname of the server, and the ownCloud setup wizard will display the following screenshot:



By logging in for the first time, an administrative account will be created. Initial setup might take a little while, after which ownCloud is ready for use.

Summary

While this chapter had nothing specific for the Cubieboard, it did teach some basic administration tasks and used them to set up some basic but useful services for a home server. While there are many more interesting services to think of, such as a DHCP server (`ics`), a printer server (`cups`), or a DNS server (`bind`), on top of that, one can build a device and incorporate it with the web server and control a light switch via a web page.

The next chapter will work on upgrading the bootloader and the kernel, two reasonably easy tasks.

6
Updating the Bootloader and Kernel

The previous chapters taught you how to get started and put the hardware and software to good use. As described in the previous chapters, software packages are updated automatically via the new packages; the bootloader and kernel, however, are not. This chapter will describe the various bootloaders available as well as the various kernels and which one to choose.

In this chapter, we will cover the following topics:

- The difference between the various bootloaders and kernels
- Obtaining a new bootloader or kernel
- Installing a new bootloader or kernel

Prerequisites for this chapter

In this chapter, you will need the destination medium used in the previous chapters, such as an SSD or a hard disk, a USB drive, and a microSD card, to boot from. Optionally, a low-capacity microSD card can also be used for this purpose instead. I have used a small 128 MB microSD card for this book, but even something on the lower side would work. All that needs to fit in the microSD card is the bootloader, some configuration files, and a kernel. About 4 MB of space would be required for that, though it will be really hard to find a microSD card with that capacity.

The bootloader overview

The bootloader is the first thing that gets loaded that is actually user modifiable. The name of the bootloader that is used by the community and throughout this book is **u-boot**. It comes in two separate flavors. First, there is the *lichee* variant but it is not actively being developed. The reason why it is still around is that it is the only bootloader that currently supports booting of the onboard NAND flash. This bootloader is nearly always preinstalled and generally there isn't any reason for it to be replaced. More interesting is the *sunxi* variant of u-boot, which is being actively developed by the community and which we will check out in the following section.

U-boot-sunxi

As precompiled versions of a bootloader are already available, this chapter is not about compiling a bootloader. They are compiled every time a new addition is done to the codebase. Each u-boot binary is unique for each board, and an entire list of bootloaders can be found at `http://dl.linux-sunxi.org/nightly/u-boot-sunxi/u-boot-sunxi/u-boot-sunxi-latest/`. There are several files, however, for each board, as follows:

- A build logfile ending with `.build.txt`
- A `sha1` hash file for the generated file ending with `.sha1`
- A file list of generated files ending within the archive ending with `.txt`
- An archive with the compiled binary files ending with `.tar.xz`

Once the correct binary file for the development board at hand is in the downloading stage, it becomes easy with the Linux utility **wget**. To use this file for Cubieboard, the following command can be used:

```
packt@PacktPublishing:~$ wget http://dl.linux-sunxi.org/nightly/u-boot-sunxi/u-boot-sunxi/u-boot-sunxi-latest/u-boot-sunxi-cubieboard.tar.xz
```

This file will then need to be extracted; tar is a common tool to do this, where the `x` parameter stands for extract, capital `J` stands for filter through `xz`, `v` asks tar to be verbose, and `f` tells tar that the next argument is a filename.

The directory name inside the archive has a complex structure appended at the end, and most likely will not be identical to the one in this example.

Let us look at the archive in more detail. In the beginning, after the u-boot identifier, there is the architecture, sunxi. Following this is the name of the board, in this case, Cubieboard. The following command line is a timestamp of when the binary was compiled. Finally, in the end, there is a hash for the specific Git commit:

```
packt@PacktPublishing:~$ tar xJvf u-boot-sunxi-cubieboard.tar.xz
u-boot-sunxi-cubieboard-20140307T104232-b5bd4c9/
u-boot-sunxi-cubieboard-20140307T104232-b5bd4c9/u-boot-sunxi-with-spl.bin
u-boot-sunxi-cubieboard-20140307T104232-b5bd4c9/u-boot.bin
u-boot-sunxi-cubieboard-20140307T104232-b5bd4c9/sunxi-spl.bin
```

Some explanation is required about these three files. When the Allwinner SoC chip boots for the first time, it cannot access the main system memory, which is also known as RAM. There is a little bit of memory, the SRAM, available inside the SoC chip. The SRAM, however, is too small to hold the entire u-boot, so u-boot is split into two: the **Secondary Program Loader** (**SPL**) and the actual u-boot. The SPL is just enough to initialize the memory and load the new, bigger u-boot into the memory. These files both need to be written to very specific regions so that the SoC understands where to boot from. To make it a little bit easier, there is a third file, called `u-boot-sunxi-with-spl.bin`. This file consists of both components combined with the appropriate spacing and is the only thing required at this moment.

Installing the bootloader

The bootloader can only be installed at one location even though the SoC can boot from three different locations. SPI-flash, at the time of writing this book, is unsupported. However, some early experiments suggest that it should be relatively easy. While u-boot could be written onto the NAND flash ROM, the community-developed version of u-boot is currently unable to read or write from the NAND flash. The only option that remains is the microSD card, which as been used until now and will be used again.

The bootloader can be installed in the following ways:

- Using a USB to microSD adapter on either a PC or on the Cubieboard
- Using an SD card reader in a PC or on the board

Be very careful when writing the bootloader. A single mistake can ruin the data on the SD card. For example, writing the bootloader at the beginning of the SD card destroys the partition table. Writing the bootloader a little too far can destroy the filesystem on the SD card. In both cases, there will be data loss because the bootloader is not in the right position, and thus the SD card is unbootable.

One requirement is that the microSD card has a partition table. It is quite common for microSD cards to have no partition table. Using the knowledge acquired in *Chapter 3*, *Installing an Operating System*, at least one partition should be created on the microSD card with at least 4 MB of free space. The partition has to be formatted either in `fat`, `ext2`, `ext3`, or `ext4`.

If no partitions from the microSD card are mounted, such as the boot partition, it is safe to remove the used microSD card and insert a different one. Depending on which device is being used, make sure you replace `sdb` with `mmcblk0` or whichever device node the SD card is assigned to. Nothing to do with bits. Copying certain files, however, might be trickier, and might have to be stored elsewhere meanwhile.

Using the `dd` tool, a new bootloader is written to `/dev/sdb`. Please make sure that this really is the device node that holds the destination SD card. Using the wrong device node can destroy important data, even rendering the system unbootable. The important parameters are `blocksize` (`bs`) and `seek`, where `blocksize`, when multiplied with `seek`, indicates the exact location on the microSD card—in this case, at exactly 8 KB. The reason is simple; this is the exact location at which the SoC chip will search the SD card for the bootloader, as illustrated by the following command:

```
packt@PacktPublishing:~$ sudo dd if=u-boot-sunxi-a10-olinuxino-lime-
20140307T10232-b5bd4c9/u-boot-sunxi-with-spl.bin of=/dev/sdb bs=1024
seek=8
```

Completing the bootloader

The bootloader does need some standard configuration files, such as the kernel that needs to be booted. A few things that cross one's thoughts are the arguments that need to be passed to the kernel and the address of the kernel to be loaded into the memory. There are three helper files that can be safely copied from the previous boot medium. They are as follows:

- `boot.scr`: This consists of a compiled list of commands that u-boot is expected to execute

- `uEnv.txt`: This consists of a list of variables that will be passed onto the kernel
- `boot.cmd`: This consists of the source file used to generate `boot.scr`

The `boot.cmd` command is not used or needed; it is just very helpful to have around. To create `boot.scr` from `boot.cmd`, the following command can be used:

```
mkimage -C none -A arm -T script -d boot.cmd boot.scr
```

Having copied these three files, the Cubieboard is almost bootable, and only the kernel remains.

Exploring the kernel

For x86-based systems, there is usually only one kernel; the one supplied by the distribution in most cases. This kernel is always based on the kernel from `http://www.kernel.org`. This kernel is often called the mainline kernel, vanilla kernel or upstream kernel. For Allwinner-based SoC chips, there is very limited support in the kernel developed by the community since 2012 with Version 3.8. There are older versions of the kernel where all the features have been added by Allwinner itself. While these changes are not of the quality that the mainline kernel accepts, it is all that there is for now. It, however, is near feature-complete and thus often the only option available to date. All these various kernels are available in various branches on a Git repository. There are nightly precompiled images available just as there were for u-boot, and they can be used in this book. Before continuing to download one of these precompiled archives, a little needs to be explained about the various versions available.

Variants of the SoC

There are various generations of Allwinner SoC chips. There is the A10, for example. The A10 series is internally known as the **sun4i**—the fourth generation of the sun series. Similarly, the **sun5i** is the internal name for the A10S, and A13. A31 and A31S are called **sun6i**. The A20 is called **sun7i,** and **sun8i** is the name for the new A23. The combined name for all these is called **sunxi**. For almost all the kernel versions and each of these machine types, there is either a Git branch or a nightly pre-compiled binary version.

Overview of the various kernels

Newcomers to the sunxi community are often confused by the various kernel versions and are not sure what kernel to use. The next few subsections explain a little about the various available versions.

Kernel Version 2.6.36

Kernel Version 2.6.36 was originally held hostage by Allwinner, which refused to release the GPL licensed code. Once it was liberated by a device maker of Allwinner-based tablets, Allwinner officially released the sources for the kernel. A community then started to form around this source release. It has been obsolete for a long time.

Kernel Versions 3.0 and stage 3.0

The Version 3.0 was the first actively developed version of the sunxi kernels. The basis for the 3.0 kernel was also initially liberated by a third-party tablet manufacturer. The community did start from scratch again and put in some heavy work into this kernel's release. Later, Allwinner also released their kernel Version 3.0, but it had already diverged compared to the community-built kernel. One major thing that was done by the community, for example, is the unification of the various machine types. Allwinner intended to release the sun4i and sun5i kernels completely separated, where the community brought it under a generic sunxi kernel and drivers. It has seen many other improvements from what was supplied by Allwinner, but not all patches and fixes were immediately put into the 3.0 series. Patches were always first put into a so-called stage tree of 3.0, and after a few weeks of testing, those patches were merged back into the 3.0 tree. The 3.0 tree is technically obsolete as well but does see some active use, as it is still in use by many tablets.

Kernel Version 3.3

Allwinner has released a new version of their kernel, but this was after the community had already started and ported everything to the mainline 3.4 kernel. The 3.3 kernel is never used by the community, as none of the bug fixes and improvements were picked up by Allwinner. It is sometimes also known as the lichee kernel.

Kernel Versions 3.4 and stage 3.4

When the 3.4 kernel was released, it was marked as the **Long Term Support** (**LTS**) release, meaning it would receive support and security patches backported to it for an extended period. The drivers for sunxi hardware were forwardported from 3.0 to 3.4 to have the Allwinner hardware support, on a well-supported kernel release. Just as with 3.0, the stage variant of 3.4 holds all the new patches and is merged to 3.4, as it is considered stable. The 3.4 series is currently being actively developed and recommended for day-to-day use.

Kernel version experimental-3.14

The highly experimental branch for Version 3.14 is not usable as of now. The idea is to take the 3.14 kernel, which is an LTS release, and apply the 3.4 sunxi drivers to it. The big difference with 3.4 is that 3.14 was a release of the kernel, and there was some actual support for sunxi SoCs. As long as these kernels are marked experimental, they should not be used except to experiment with.

The devel branch of the kernel

Patches that are written against the mainline kernel end up on various mailing lists while they are being reviewed. The devel branch, which stands for development, tries to capture all these under-review patches and merges them into the mainline following tree. In other words, this tree holds support for most of the hardware to this date, but it may very well change as the review process continues. This kernel should only be used when working on drivers, doing reviews, or wanting to help testing new things. This branch, however, may be removed in future as things start to settle.

Next branch of the kernel

Very similar to the devel branch, it tracks the mainline kernel and holds accepted patches. It could be considered as the stable variant of the devel kernel. This kernel should only be used when there is a need for the next and upcoming kernel release.

Choosing a kernel

With so many kernels available, there really is only one kernel of interest at this moment—the 3.4 kernel. While this may change when 3.14 leave the experimental state and start to replace 3.4—for now, they are not an option. With that choice set, there's the machine type to pick from, depending on the type of Cubieboard being used. The variants section discussed earlier will explain which machine type to go for.

Installing the kernel

Using `wget`, an A20-based kernel will be downloaded and extracted in the following example:

```
packt@PacktPublishing:~$ wget http://dl.linux-sunxi.org/nightly/linux-
sunxi/linux-sunxi-3.4-sun7i/linux-sunxi-3.4-sun7i-latest.tar.xz
linux-sunxi-3.4-sun7i-20140215T115414-8ea347b/lib/
linux-sunxi-3.4-sun7i-20140215T115414-8ea347b/boot/
```

Also here, a new directory containing a `datestamped` and `githash` named directory will be created. Within this directory, there will be two subdirectories. The boot directory holds the actual kernel, a file called **uImage**. It is to be placed on the first partition of the microSD card.

Installing the kernel modules

In a kernel, drivers can be compiled into the kernel and are always loaded, or they are separated from the kernel and can be loaded as required. These separated drivers are called modules and on most distributions live at `/lib/modules/`. Copying over this directory copies all the files needed for this new kernel.

The following command assumes that this is done natively on the Cubieboard. When using a different system to perform the copy, the destination path will be different:

```
packt@PacktPublishing:~$ sudo cp -ar linux-sunxi-3.4-
sun7i-20140215T115414-8ea347b/lib /
```

With both the kernel and modules in place, the system can be rebooted safely using the new kernel and its modules.

Summary

This chapter covered the various bootloaders and kernels and briefly explained the differences between them. It then showed you how to install a precompiled kernel onto a microSD card to be used as a boot device.

The next chapter will go over the steps of actually compiling a bootloader or kernel from scratch using sunxi-bsp or the sunxi board.

7
Compiling the Bootloader and Kernel Using a BSP

Sometimes, a special feature in the kernel is required that is not included in the precompiled binaries, or maybe some new piece of Allwinner-based hardware was obtained that is not yet supported in the existing list of precompiled files. To solve issues like these, you need to compile the bootloader or kernel from source. While it is perfectly possible to download and compile the bootloader and the kernel by itself, the linux-sunxi community developed a **board-support-package** (**BSP**) that allows you to compile all these components together.

This chapter will cover the following topics:

- Installing a toolchain
- Obtaining and using the BSP
- Compiling the bootloader
- Compiling the kernel
- Creating a hardware pack

Prerequisites

Compiling things requires a compiler toolchain. Here, there are two options. Either compile on Cubieboard itself or cross compile on a regular PC. Getting a functional toolchain working on anything but Linux is up the reader to solve. The options are thus to use the installation created in the previous chapters and compile directly on Cubieboard or to have a (virtual) Linux machine available where a cross-compiler can be installed and used. In this chapter, both the methods will be described. Additionally, a working Internet connection on the device that is being used to compile on is initially required; this is to obtain the source code.

Installing a toolchain

A toolchain is a collection of tools, including a compiler, required to compile the source code into binary forms. In theory, just the compiler is enough, but various other bits and pieces are often used to help with the compilation. One such example might be well known, the `make` command. A combination of all the tools required to compile things is known as a toolchain. Installing a toolchain varies from distribution to distribution; we will only cover a few examples here.

Debian or Ubuntu

For Debian, the toolchain is called `build-essential`, and when cross-compiling, the arm compiler needs to be installed on top of that; this package is called `gcc-arm-none-eabi`. Unfortunately, at the time of writing this book, the `gcc-arm-none-eabi` package did not exist in Debian wheezy. The version to be released after wheezy is jessie, which contains this cross compiler. One way to get it is to set up a virtual machine and install the testing version of Debian there.

Fedora

On Fedora, the installation is slightly different. Here, the `groupinstall` variant needs to be used to install the `Development Tools` and `Development Libraries` packages. To cross compile, install the `gcc-arm-linux-gnu` package in addition to that. It is necessary to use double quotes ("") due to the spaces in the meta packages.

Other distributions

Fedora and Debian are of course only two distributions out of the potential hundreds. These two distributions, however, give you a very good indication of how most other distributions handle the installation of the toolchain. These two significant distributions do it differently. For the Gentoo distribution, there is a crossdev toolset, which will compile and install a cross-compiler toolchain. After installing crossdev, use `crossdev --target arm-pc-linux-gnueabi` to install the arm toolchain. The arch distribution has the `gcc-arm-linux-gnueabihf-bin` package in the **Arch User Repository** (**AUR**).

There are also vendor-supplied toolchains, such as the ones supplied by Linaro and CodeSourcery, for example. These are to be manually downloaded from the vendor site. Often, the toolchain comes in a tarball or a ZIP package and needs to be manually extracted. These toolchains are commonly used when no native arm-toolchain is available. Linaro even offers their cross-compiler for OSX and Windows; however, these two require an immense amount of work before you can start compiling.

Other required tools and packages

Having obtained a complete toolchain, a few other packages are still required. Git is a tool used for source-code management; the package is named `git` on almost all distributions.

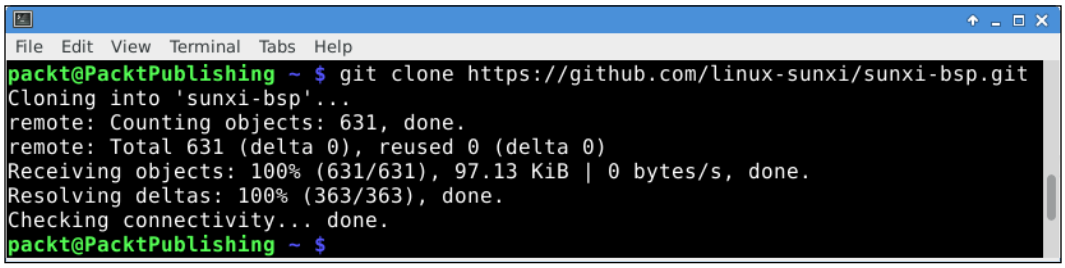
Additionally, `u-boot-tools` is required. The package is also named `u-boot-tools` or `uboot-mkimage` on some distributions.

On some distributions, when installing the toolchain, as described earlier, the **pkg-config** package sometimes doesn't get installed and needs to be installed on the distributions that lack it. The package is nearly always called pkg-config. To compile some of the tools, the `libusb` header files are required. The package name can vary between distributions. On Fedora, it is called **libusb-devel**. On Debian and Ubuntu, it is called **libusb-1.0-0-dev**. Please note that `libusb` is often available under several versions in many distributions. The version required is `1.0`, and other versions may cause the compilation to fail.

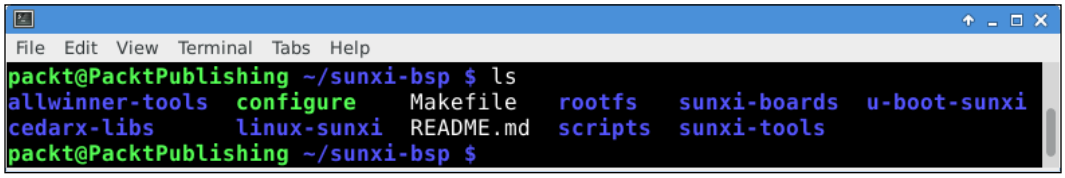
Finally, the *ncurses* header files and libraries are required; the package is called either **ncurses-dev** or **ncurses-devel**.

Obtaining and maintaining the BSP

Whether the BSP is going to get cross-compiled or natively compiled, obtaining and using it is identical, and thus the instructions are common. All the code are stored on a git-server, and GitHub or Gitorious can and should be used as the main mirrors to obtain them from. Using Git, the repository can be cloned from one of the mirrors, as shown in the following screenshot:



After entering sunxi-bsp, the following list of files and directories will be visible:



Let us take a minute to quickly go over this list of file directories, of which some are actually separate Git repositories:

- `allwinner-tools`: This is a collection of files, drivers, and tools when working with the Allwinner-supplied material, such as livesuit. It is not of importance when working with the community tools.
- `rootfs`: These are the files to be placed into the generated root filesystem, commonly named `rootfs`.
- `sunxi-boards`: These are the FEX files of the community-supported boards. Refer to *Appendix C, The FEX Configuration File*, for more information about FEX.
- `u-boot-sunxi`: This is the community-developed bootloader.
- `cedarx-libs`: These are proprietary libraries for the **Video Processing Unit** (**VPU**) supplied by Allwinner.
- `linux-sunxi`: This is the community-developed Linux kernel.
- `scripts`: These are various scripts used by BSP.

- `sunxi-tools`: These are community-developed tools to work with Allwinner hardware, including the FEX to a binary `script.bin` compiler.
- `Makefile`: This is a file to control how to compile various repositories.
- `Configure`: This is a script to choose and configure the entire BSP.
- `README.md`: This is a simple text file with some basic usage instructions.

Updating the repositories

Some of the directories within the BSP are not yet populated. The BSP is smart enough to populate the required repositories by itself as and when it needs them. If, however, one of the repositories is to be manually populated or more importantly, updated, Git can be used to do so. Adding the `–init` parameter after the update is required when updating the repository for the first time, as follows:

```
packt@PacktPublishing:~/sunxi-bsp$ git submodule update --init sunxi-
tools
```

Omitting the last parameter, in this case, `sunxi-tools`, will update and populate all the Git subrepositories. However, this will not always yield the latest version of the repository. The BSP controls the version of each repository to use. It can be said that it locks each of the subrepositories to a certain version. The BSP itself can be updated using Git, as follows:

```
packt@PacktPublishing:~/sunxi-bsp$ git pull
Already up-to-date.
```

If, however, the BSP is not updated or does not include the latest updates to subrepositories, the subrepositories can be manually updated. To update one of the repositories, enter it, and use the regular Git commands to update or check out a different branch as follows:

```
packt@PacktPublishing:~/sunxi-bsp$ cd sunxi-tools/
packt@PacktPublishing:~/sunxi-bsp/sunxi-tools$ git pull
```

Do note that this could potentially break the version control of the BSP itself. Or rather, the local BSP will no longer match the official BSP. To delete all changes made to the local subrepository and bring the BSP in sync with the upstream version, the following command can be used:

```
packt@PacktPublishing:~/sunxi-bsp$ git checkout – sunxi-tools
```

Updating and modifying the submodules in Git is perfectly safe and is done frequently by developers. Do be careful when getting started and even more so when unfamiliar with Git.

Some experience with version control systems, especially Git, is strongly recommended before tinkering with the repositories. In case all else fails, feel assured that it is always possible to remove the BSP and start again.

Choosing a kernel

As discussed in *Chapter 6*, *Updating the Bootloader and Kernel*, there are a few different kernels available. These kernels are built from the various branches available in the Git repository. Listing these branches is done using the `git branch` command, with the addition of the `-a` parameter telling Git to show all the available branches. In the following screenshot, the kernels discussed in *Chapter 6*, *Updating the Bootloader and Kernel*, should be recognizable:

```
File  Edit  View  Terminal  Tabs  Help
packt@PacktPublishing ~/sunxi-bsp $ cd linux-sunxi/
packt@PacktPublishing ~/sunxi-bsp/linux-sunxi $ git branch -a
* (detached from 9ee9fc5)
  sunxi-3.4
  remotes/origin/HEAD -> origin/sunxi-3.4
  remotes/origin/experimental/sunxi-3.10
  remotes/origin/experimental/sunxi-3.14
  remotes/origin/experimental/sunxi-3.14-android
  remotes/origin/lichee-3.0.8-sun4i
  remotes/origin/lichee-3.0.8-sun4i-an7g3
  remotes/origin/lichee-dev
  remotes/origin/mirror/allwinner-2.6.36
  remotes/origin/mirror/android-2.6.36
  remotes/origin/mirror/android-3.0
  remotes/origin/mirror/android-3.10
  remotes/origin/mirror/android-3.3
  remotes/origin/mirror/android-3.4
  remotes/origin/mirror/master
  remotes/origin/reference-3.0
  remotes/origin/reference-3.10
  remotes/origin/reference-3.14
  remotes/origin/reference-3.4
  remotes/origin/stage/sunxi-3.4
  remotes/origin/sunxi-2.6.36
  remotes/origin/sunxi-3.0
  remotes/origin/sunxi-3.4
  remotes/origin/sunxi-devel
  remotes/origin/sunxi-next
  remotes/origin/wip/cleanups-linux-3.3
  remotes/origin/wip/experimental/sunxi-3.10
  remotes/origin/wip/lichee3-sunxi/import-a10s-sdk
  remotes/origin/wip/lichee3-sunxi/import-sun5i
  remotes/origin/wip/linux-sunxi-3.0/mem
  remotes/origin/wip/linux-sunxi-3.0/mem_mali
  remotes/origin/wip/linux-sunxi-3.0/sun5i
  remotes/origin/wip/linux-sunxi-3.4/cubieboard
  remotes/origin/wip/stage-sunxi-3.4/a20
  remotes/origin/wip/sunxi-3.4/rtl8188eu
packt@PacktPublishing ~/sunxi-bsp/linux-sunxi $
```

The detached branch, in this case, is the kernel version that is linked to the BSP at the time of writing this book. Using `git checkout`, it is easy to switch to an alternative branch and eventually to a kernel. This can be seen in the following screenshot:

Compiling for a Cubieboard

Before compiling for a Cubieboard, the BSP has to be configured first. Whenever building for a different development board, the BSP will have to be reconfigured. This however, is an easy task using the configure script. Running configure without a parameter will populate the `sunxi-boards` repository, as that repository contains a list of supported boards and prints a list of available boards, as shown in the following code. Take note of the prefix `./`, which is used to the configure the script. The output generated by configure is reduced here for clarity and convenience:

```
packt@PacktPublishing:~/sunxi-bsp$ ./configure
Usage: ./configure <board>
supported boards:
        * a10s-olinuxino-m a10s-olinuxino-m-android
        * a10-olinuxino-lime a10-olinuxino-lime-android
        * a13-olinuxino a13-olinuxino-android
        * a13-olinuxinom a13-olinuxinom-android
        * a20-olinuxino_micro a20-olinuxino_micro-android
        * cubieboard cubieboard-android
        * cubieboard2 cubieboard2-android
        * cubietruck cubietruck-android
```

There are two variants for each board that are to be passed to the configure script; the Android variant is specifically used to build Android kernels. While Android is Linux, there are some small differences that need to be accounted for. In the following example, the BSP is configured to build for Cubieboard2:

```
packt@PacktPublishing:~/sunxi-bsp$ ./configure cubieboard2
cubieboard2 configured. Now run `make`
```

Now, before running, as suggested by the BSP, one more thing needs to be mentioned. The BSP wants to know what compiler to use, and it knows this from the `CROSS_COMPILE=` parameter. By default, this parameter is forced to `arm-linux-gnueabihf-` via the Makefile, which is the prefix to the installed arm compiler. Thus `gcc` is expected to be named `arm-linux-gnueabihf-gcc`. Things get more interesting when compiling natively on one of these boards. This is because in theory, no cross-compiler is desired, `gcc` should just be called gcc. To remedy this, pass an empty `CROSS_COMPILE=` parameter to make, as follows:

```
packt@PacktPublishing:~/sunxi-bsp$ make CROSS_COMPILE=
```

Otherwise, the installed compiler prefix needs to be added, and yes, the dash at the end is a part of the prefix. If you are compiling on Debian, the following command can be used:

```
packt@PacktPublishing:~/sunxi-bsp$ make CROSS_COMPILE=arm-none-eabi-
```

Every distribution tends to name their cross-compiler differently; there is no right or wrong. Using arm- and double tab completion should yield the prefix for almost all distributions. Also, adding the correct cross-compiler prefix to the Makefile can be very helpful here.

Depending on the amount of memory and which system is used to perform the compilation, this could take from several minutes to an hour or two! If there are strange crashes or problems, before looking at *Appendix A*, *Getting Help and Other Helpful Online Resources*, about contacting the community for support, make sure that the chosen board is properly supported and supplied with adequate power. Quite often, overclocked memory or a lack of enough power will show up under the heavy stresses that a kernel compilation encompasses.

While it is nice to be able to compile the standard kernel, often, someone will want to compile a kernel due to customization, for example, with certain drivers or options added or removed. Even a custom patch is something that needs a custom-compiled kernel. Normally, the `menuconfig` command is used in a kernel directory. The BSP also allows this by passing the `linux-config` parameter to the `make` command, as follows:

```
packt@PacktPublishing:~/sunxi-bsp$ make linux-config
CROSS_COMPILE=arm-none-eabi-
```

Following this will yield a standard `menuconfig` session.

Some other parameters to the `make` command are Linux or u-boot, which are used to compile only the Linux kernel or only u-boot. The resulting binaries will be located under the `sunxi-bsp/build` directory under their own respective trees.

When the compilation is completed, a so-called hwpack or hardware pack is created in the `sunxi-bsp/output` directory. The hardware pack is an archive containing three subdirectories. The first is called `bootloader` and contains the `u-boot-sunxi-with-spl.bin` bootloader.

In the `kernel` subdirectory, the board-specific kernel, named `uImage`, lives combined with the board specific `script.bin` file.

The final directory is the `rootfs` directory. This directory contains everything specific for the chosen target board. The content can and should be copied to the target root filesystem.

Installing the files in the hwpack, specifically the bootloader, was described well in the *Chapter 6, Updating the Bootloader and Kernel*.

Summary

In this chapter, the basics of the BSP were covered. Using the BSP in combination with Git, which is a powerful tool to download and manage the various source repositories, you can compile various components and generate an easy-to-use device-specific hardware pack.

The next chapter will cover **general purpose input output pins** (**GPIOs**). This can be useful to do various jobs, including blinking an LED!

8
Blinking Lights and Sensing the World

Until now, we have covered several aspects of Cubieboard, but they were all software-related. When interfacing with real-world scenarios, ranging from blinking a **Light Emitting Diode** (**LED**) and switching on a light to spinning motors, things become much more interesting. There are numerous things you can do once you connect the board to the various devices available today. A robot vacuum cleaner very much started its development life on a development board, where engineers connected various sensors to make the robot see what was around it. Connecting all these devices can be highly complex and might involve more than just the basics of electronics; therefore, this chapter will be dedicated to the fundamentals of electronics to help you to get to grips with some basics in connecting an LED, making it blink, and connecting a button to respond to a push.

In this chapter, we will cover the following topics:

- An overview of a few concepts in basic electronics
- How to toggle a **General Purpose Input/Output** (**GPIO**)
- How to connect a button and read its status

Making an LED glow

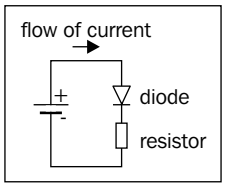
A world without LEDs seems almost unimaginable today. They inform us of messages on our smartphones, they have been used as indicator lights on TV and stereo equipment for years, and there are even big screens built of nothing but LEDs. Ironically, the invention of the LED was more of an annoying side effect of the first diodes. The intention was not to emit light at all, and in early equipment, the diodes were covered in black paint to hide their glow. The name LED is so ubiquitous that their meaning might almost have been forgotten.

Making an LED glow, however, does require a few tricks and some know-how to make sure not to break it. Each LED has certain characteristics, such as the amount of voltage it requires to function and the maximum amount of current that can flow through it. A blue LED, for example, requires more voltage and more current than a red LED. It is, therefore, imperative to know the technical details of an LED.

Resistance required

An LED is just a special form of diode. It is a one-way street for electricity, that is, the current is only allowed to travel in one predefined way. Also, caution must be taken while connecting an LED; if it is connected in the wrong direction, no current will flow at all, thus emitting no light. The other way around, however, is more interesting.

Let us examine what happens when an LED is connected in the correct direction. Once a certain threshold voltage is supplied, read the forward voltage, the diode starts conducting, and the LED emits light. As for the current flowing through the diode, things look a little different. A diode will conduct all the current that is supplied to it. Different kinds of diodes serve different purposes in the electric design that they are used; in the case of LEDs, the purpose is often to emit light. These types of diodes often can only sustain a very small amount of current to flow through them, usually in the milliampere range. A resistor, as the name suggests, restricts the flow of the current going through it and thus is used for regulation of current, as depicted in the following diagram:



There are plenty websites that can help in calculating the correct resistance for a given voltage and current; the math behind it is not very complex. Ohm's law is the common formula used to calculate resistances. It is represented as follows:

$$R = \frac{V}{I}$$

And:

$$R = \frac{V_s - V_f}{I}$$

To calculate the resistance, R, the voltage, V, needs to be divided by the current I. Let us understand how we determine the required voltage. For this, the diode's forward voltage is subtracted from the available voltage. This produces the following variant of Ohm's law:

$$60 = \frac{3.0 - 1.2}{0.030}$$

Let us assume that our voltage source is 3.0 volt and the forward voltage of the diode is 1.2 volt. Suppose the required current for this example is 30 milliampere, then filling in these values in Ohm's law yields the following result:

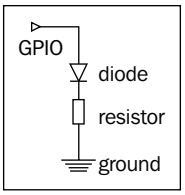
From this little bit of math, we understand that a resistance of 60 Ohm is required to make the LED glow without trying to consume all the available current. A resistor of 60 Ohm, however, might be hard to find, and only two options are available. Connect several resistors in series, one after another, to obtain the required value, or find a resistor that is reasonably close in value, such as 68 Ohm. Rearranging Ohm's law and filling it in yields 26.5 milliampere of current flowing through the LED. This should generally be enough to light up the LED. This calculation can be seen in the following formula:

$$0.0265 = \frac{68}{3.0 - 1.2}$$

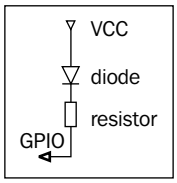
Using a resistor to limit the current flowing through an LED is not the most power-efficient method of lighting up an LED, but is very common and the simplest method. Ideally, a constant current supply is recommended, which would no longer require a resistor. After all, the maximum available current to the LED is what it would like to receive in the first place. While out of the scope of this book, it is a good exercise, to learn more about constant current power supplies and driving LEDs.

Sinking and sourcing

To connect a device to a GPIO, it would be ideal if the GPIO can deliver the correct voltage and current. Unfortunately, this rarely is the case. The voltage supplied by a pin is often either the supply voltage or the I/O voltage, both of which are input voltages to the chip. In the case of the Allwinner A series of chips, this will be 3.3 volt. The amount of current a GPIO can deliver or absorb is limited. Supplying current, or sourcing, as it is called, can vary from 10 milliampere to 40 milliampere. The amount of current a pin can source is configured in the FEX file, which is explained in more detail in *Appendix C*, *The FEX Configuration File*. The opposite of sourcing is sinking, but before that is covered, let us first examine the following diagram where an LED is connected directly to a GPIO:



In the preceding diagram, it is quite obvious that the GPIO pin supplies the voltage and the current flows through the diode into the ground. The GPIO is the "source" of the current. The opposite is possible too; the GPIO absorbs or sinks the current, as shown in the following diagram:

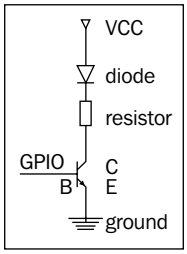


In the preceding diagram, the diode gets supplied with a voltage and current from a power source; in electronics, it is often called VCC. The current then flows as before through the diode and through the resistor, and is then absorbed by the GPIO. The GPIO "sinks" the current. While this second method works just as well as the first, there are a few things to notice. The power source needs to supply an appropriate voltage, or the GPIO can get overloaded and burn up parts or even the whole chip. On the software side, the pin will function in the opposite way; when the GPIO is high, the voltage is the same as that of VCC and thus no current flows. If the pin is set to low, the voltage can now flow.

An observant reader will probably notice that if the GPIO is set up to a source of 30 milliampere, a resistor is not even needed. While it is bad practice, it is still possible; however, it should never be done in a final design.

Amplifying the voltage and current

All the preceding methods work only if a small amount of current is required. What do you do when there is a higher current demanded? In this case, a simple transistor is used. A transistor can be thought of like a valve on a water pipe. Normally, the valve is closed and no water is allowed to flow. When current is applied to the base **B** of the transistor, the valve slowly starts to open and conduct current from the collector **C** to the emitter **E**. The flow of current is proportional to the current applied to the base, as a transistor is by nature a current amplifier. In addition to that, the current applied to the base is also added to the output of the emitter, as shown in the following diagram:



The voltage supplied via the VCC is determined by the specification of the transistor, not the GPIO. It is easily possible that 24 volt is regulated via the transistor, assuming the transistor allows this, while controlling it via the 3.3 volt of the GPIO. In essence, this is a voltage amplifier, and a transistor is most commonly used for this effect.

There are two kinds of transistors: PNP and NPN. The general working of an NPN transistor was explained in the previous paragraph. The PNP transistor works the other way around, meaning it conducts current by default when there is no current applied to the base of the transistor via the GPIO. If current is applied to the transistors' base, the transistor will stop conducting current.

Controlling pins from software

There are several ways to control a GPIO from within Linux. From within the kernel through a driver or from one of the many programming languages, such as a C program or a Python script. The quickest and most basic way, however, is right from the console. This is assuming, of course, that the driver to control the GPIOs is loaded as a module or compiled into the kernel. Refer to *Chapter 7*, *Compiling the Bootloader and Kernel Using a BSP*, to recompile a kernel and to add the GPIO driver. Depending on the kernel version used, it is called the *GPIO support for the sunxi platform* and can be found under **GPIO** in the **Device Drivers** option. Depending on which GPIO is used, the FEX file will need to be modified to configure the GPIO pin. To set up one pin as output GPIO, the following code snippet can be used:

```
[gpio_para]
gpio_used = 1
gpio_num = 1
gpio_pin_1 = port:PB03<1><1><default><default><default>
```

If everything has been set up appropriately, the GPIO pins can be toggled via the filesystem, for example, writing `1` into `pb03` makes the GPIO high, as shown here:

```
packt@PacktPublishing:~ $ echo 1 > /sys/devices/virtual/misc/sunxi-
gpio/pin/pb03
```

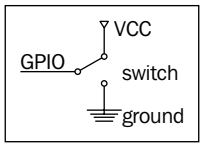
Similarly, writing `0` makes it low again.

Not only is a transistor required on higher current loads from one pin, often there is a maximum current that can be sourced from all the combined GPIOs. Thus, the best method to control things is to always use a transistor.

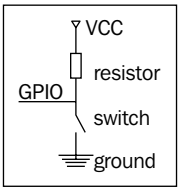
Pulling up and pulling down

A GPIO that is configured as an input simply takes a measurement of the voltage applied to it. If it is above a certain threshold, it is considered to be high, or `1`. If it is below a certain threshold, it is considered as `0`. In the case of the Allwinner A series, generally speaking, everything above 2.5 volt is considered high, and everything below 0.8 volt is `0`. Anything in between is undefined and could be either `0` or `1`.

With that knowledge at hand, it is easy to just have a three-way switch and connect it to either the ground or the 3.3 volt. The following diagram illustrates a switch directly connected to a GPIO:



There are a few caveats to consider. There is a transition phase, where the switch moves from one state to the other and the voltage could be anything between 3.3 volt and 0 volt. Additionally, the maximum amount of current that a pin can source will flow into the GPIO, making a very inefficient design and could even damage the chip. Both issues can be addressed by connecting either state permanently to the GPIO via a resistor. If a resistor is used to permanently connect the VCC, it is called a pull-up resistor, as the resistor pulls the voltage up. If the resistor is used to pull the pin down to ground, it is called a pull-down resistor, as the resistor pulls the voltage down to ground. The following diagram demonstrates the usage of a pull-up resistor in conjunction with a switch:



In this scenario, the GPIO will see the high voltage and interpret it as a logical `1`. When the switch is pressed, the current prefers to take the path of least resistance, and as the GPIO has a natural resistance internally, the ground is where all the current will go.

If the positions of the switch and the resistor are swapped, a pull-down circuit is created. In such a scenario, by default the GPIO will see the low voltage and interpret it as a logical `0`. When the switch is pressed, the current will naturally flow to the ground and the GPIO interprets it as a logical `1`.

In both cases, the resistor value needs to be calculated in much the same way as the resistor for the LED was. Ideally, the resistor will be as high as possible so that only a very limited amount of current can flow, thus not wasting any power. Choosing a resistor that is too high, however, results in too little current to flow, and thus the pin cannot sense the voltage.

In both these cases of the Allwinner SoC, there is an internal pull-up or pull-down resistor available that can be configured via the FEX file. The following command can be used in this case:

```
[gpio_para]
gpio_used = 1
gpio_num = 1
gpio_pin_1 = port:PB03<1><0><1><default><default>
```

For additional details, see *Appendix C*, *The FEX Configuration File*. The choice of whether to use a pull-up resistor or a pull-down resistor depends on the project's requirements. Generally speaking, if the pin has to be read as `0` by default and changed to `1` when a button is pressed, a pull-down resistor is used. In practice, however, a pull-up resistor tends to be more power efficient and is commonly used.

Reading a switch

Reading the status of a switch is similar to writing to the LED. All the prerequisites apply equally. The only difference is that the GPIO is read via `cat`, rather than written via `echo`, as shown here:

```
packt@PacktPublishing:~ $ cat /sys/devices/virtual/misc/sunxi-
gpio/pin/pb03
1
```

Using this knowledge, reading a button can be done from various programming languages in various forms.

A good exercise can be to write a simple script that reads the status of the button and makes the LED reflect this status. Both of the previous examples use the same pin as GPIO, so a second pin will need to be configured as GPIO if not already configured.

From here on out, there are many more ways and methods to interact. There are **analog-to-digital converters** (**ADCs**) available to read an analog voltage, the **Serial Peripheral Interface** (**SPI**) bus and the **Inter-IC** (**I2C**) bus, also known as the **Two Wire Interface** (**TWI**), to connect a plethora of peripherals, each allowing wonderful inventions to be created. Since this is an introductory book, going into detail would require many more pages explaining the various techniques. But hopefully, an interest has been sparked in you to find resources to continue experimenting and working with Allwinner-based boards.

Summary

In this chapter, we covered some fundamental electronics, teaching the reader how to connect an LED and a switch. We also covered the most common pitfalls and best practices when interfacing with LED switches. Blinking an LED is considered as the Hello World of embedded development.

A

Getting Help and Finding Other Helpful Online Resources

Having worked through this book, there will probably be many questions that are still left or even after consulting *Appendix D*, *Troubleshooting the Common Pitfalls*, it might be quite possible that something just does not work out. Luckily, there are quite a few helpful volunteers who are willing to help.

This appendix will cover the following topics:

- Meeting various communities
- Where to get help
- How to ask the right questions
- Getting a new Allwinner-based device supported

Meeting the community

It should come as no surprise that there are many online communities surrounding various projects and products. This is especially true considering all the development boards covered in this book and their SoCs. Each of these communities more or less focus on certain tasks. There are three most important ones with regard to this book. Firstly, the Olimex community, which focuses on all Olimex-created products and puts forward questions related to Olimex boards and peripherals. Similarly, there is a Cubietech community, which revolves around Cubieboards. Lastly, there is another important group of volunteers well-known among the bunch, and that is the linux-sunxi community. The name stems, as mentioned earlier, from the Allwinner SoC family and also pertains to their main interest. The linux-sunxi community focuses on bootloader, kernel, and driver development and gathers a lot of information about Allwinner SoCs on their wiki.

While it is wise to get help or, even better, to give help to other users via the Olimex or Cubietech communities when working on their hardware, the linux-sunxi community is very knowledgeable and generally helpful.

None of these communities have fulltime paid employees employed by any of these companies. They are volunteers who do not mind helping other users as they themselves might have received help from others. Additionally, this community has people from every corner of the world, so make sure you keep things nice and polite.

Getting in touch with the Olimex community

The main contact point for Olimex is obviously their website, `http://www.olimex.com`. While the site's initial and main focus is their shop, they have links to reach their forum, their wiki, and even to projects done by users to inspire others. The direct wiki URL is `http://www.olimex.com/wiki`, and the direct URL for the forum is `http://www.olimex.com/forum`, which can be used to bypass the main site if desired. An account for the wiki will need to be requested with Olimex, but anybody can create a free forum account and get support there. Additionally, there is an **Internet Relay Chat** (**IRC**) channel on the Freenode IRC network, called #*olimex* where live chat is possible with other community members.

Getting in touch with the Cubietech community

The official website of Cubietech is `http://www.cubieboard.org`. The main site is mostly a blog where new products are announced. While the site suggests that there are various forums, these are actually links to unofficial external forums in various languages. Do note that this can be both an advantage as one can communicate in their native language, and a disadvantage as resources are split among various forums. Cubietech does not have an official forum, but rather uses a Google-groups mailing list that can be accessed as a forum via the Google web interface. The direct link to their forums is `http://www.cubieboard.org/forum/`. There is also a read-only wiki available called **docs**, which can be directly accessed via `http://docs.cubieboard.org`. Also, Cubietech has an IRC chatroom on the Freenode network called #*cubieboard*. Finally, Cubietech has a web page, `http://www.cubieboard.org/support/`, where this and more is mentioned how to get support.

Getting in touch with the linux-sunxi community

The linux-sunxi community also has an official website, which is their very resourceful wiki and can be found at `http://linux-sunxi.org`. The wiki itself can be quite daunting but also very helpful. There are many tutorials and *how-to* sections for various tasks. Content is being continuously worked on by kernel hackers and regular users alike, including the author of the book. The linux-sunxi wiki might be a little more technical than this book, but it is a great resource that should not be overlooked. Additionally, there is also a mailing list where a lot of communication happens. Instructions on how to join or just browse the mailing list can be found on the linux-sunxi community website at `http://linux-sunxi.org/Mailing_list`. Just as with Olimex and Cubietech also, the linux-sunxi community can be found on the Freenode IRC network on channel #*linux-sunxi* where many users and developers contribute their knowledge.

Getting help by asking the right questions

While the *where* has been covered in the earlier sections, the *how* is an equally, if not more, important aspect of getting help. Community members often do want to help, but questions such as *Help me, it doesn't work* are wrong on various levels; for one, what does not work? What have you tried so far to make it better? Also, it sounds demanding and when asking for help from a volunteer, demanding things usually does not help at all. So when asking a question via a forum, a mailing list, or an IRC channel, always try being polite and as clear as possible on how things are going wrong. Do not let language be a barrier; many community volunteers do not speak English natively themselves and understand that you might not be a native English speaker either, and that is okay. If in doubt or unsure at any point, Wikipedia has a small entry about how to behave in the online communities where the netiquette and online etiquette sections are of interest; you can find this at `http://wikipedia.org/wiki/Etiquette_in_technology`. Additionally, Eric S. Raymond also posted an interesting read called *How to ask questions the smart way* at `http://www.catb.org/~esr/faqs/smart-questions.html`.

Getting support for any new Allwinner-based hardware

Allwinner SoCs are widely used, and while some devices are very well documented, such as the Cubieboard, others might not yet be. Whether you have found a new device or are developing a new device based on one of the many development boards, this new device might not be supported yet. No worries, as Allwinner-based devices are extremely similar to each other, and all the knowledge acquired by working through this book will still be useful. But the hardware in question might not yet be supported by the kernel, the community, or otherwise. The linux-sunxi community has developed a *New Device howto* on their wiki that can be used to add support on your own! With the knowledge learned from this book at hand and from referring to the website, `http://linux-sunxi.org/New_Device_howto`, this should be no problem at all.

Summary

This appendix has introduced you to the main communities where additional help can be acquired from volunteers. The next appendix will give an overview of the various basic Linux commands used in this book.

B

Basic Linux Commands Cheatsheet

A Linux-based system contains many commands. Each installed application is, in fact, a command. This appendix will give an overview of the most basic commands that should, in theory, be available in every basic Linux installation. Most of these commands will be used throughout this book, some of which are considered as the bare basics. This appendix is by no means an authoritative reference, but it should get you well on your way.

Requesting the manual

Linux features an interesting command called `man`. It is special because, if installed, it opens a manual page about any command. Try it by requesting the manual page of `man`, as follows:

```
packt@PacktPublishing:~$ man man
```

Running `man` with the `man` parameter will open the manual page of `man`. With the *q* key, you can exit `man`, and with the arrow keys, you can navigate around. The *h* key opens a help screen where more keys are explained.

If `man` or manual pages are not installed, the Internet can be used instead. There are many sites that have the most common manual pages available. The website `http://www.die.net` is very popular and can be used to query various manual pages.

Something to note is that there are several sections of manual pages available—nine to be exact. The man-manual page will explain each of them briefly. The first section relates to commands and these pages are queried by default.

Finally, a lot of commands often have a short help screen, which can be activated by appending `-h` or `--help` to a command.

Listing a directory

The `ls` command, which stands for list, is the command used to list the contents of a directory. Without a parameter, it will list the current active directory, and if supplied with a parameter, it will try to list that file or directory, as follows:

```
packt@PacktPublishing:~$ ls /home/
packt
```

Changing through directories

To change to a different directory, the `cd` command can be used. Without a parameter, `cd` will always change to the current user's home directory; otherwise, the directory that is supplied via the first parameter is used, as follows:

```
packt@PacktPublishing:~$ cd /home
packt@PacktPublishing:/home$
```

Getting the current working directory

The current active directory or current working directory can be printed using the `pwd` command. This can be useful when one wants to know where one is located in the current filesystem and can be done using the following command:

```
packt@PacktPublishing:~$ pwd
/home/packt
```

Getting the current user

Finding out which user is currently logged in can be useful, especially when swapping between several users. The `whoami` command will print the current active logged-in user, as follows:

```
packt@PacktPublishing:~$ whoami
packt
```

Running commands as root

Very often, when administering or setting up a system, certain commands need to be executed as root. The `sudo` command, when set up properly, can be used to allow certain users to execute certain commands as root. The *who* and *what* queries are controlled via the `sudoers` file at `/etc/sudoers` and should be edited with the `visudo` command. The `sudo` command is used as a prefix to the command to be executed as root, as follows:

```
packt@PacktPublishing:~$ sudo whoami
[sudo] password for packt:
root
```

It should be noted that while `sudo` is very often used to execute commands as root, it can also be used to have any user execute any command as any user.

Changing the current user without logging out

To actually change to a different user as if one would log in with that user, the `su` command is used. With `su` followed by a different username, it is possible to change the identity of the said user. Unlike `sudo`, which requires the current user's password, here, the user to whom access is being requested is required, as shown in the following command:

```
packt@PacktPublishing:~$ su root
Password:
```

Logging out of a shell or from a different user, the following `exit` command is used. It takes no parameters. Alternatively, *Ctrl* + *d* can also be used to log out on nearly all shells.

```
packt@PacktPublishing:~$ exit
```

Creating files or changing their dates

To create a new empty file, the `touch` command can be used. Additionally, to modify an existing file's access and modification date can be changed to reflect a new date and time, as follows:

```
packt@PacktPublishing:~$ touch /tmp/testfile
```

Creating directories

To create a new empty directory, the following `mkdir` command can be used:

```
packt@PacktPublishing:~$ mkdir /tmp/testdir
```

Removing files

To remove a file, the `rm` command can be used. The `rm` command removes the file that is passed along as a parameter. By default, `rm` will refuse to remove a directory; it only operates on files.

The two options that are very often passed to `rm` are `-r` and `-f`. First, a word of caution on the `-f` option, which stands for force; while the `rm` command should be used with extreme care, the `-f` option requires even more thought and attention. The `-f` option forces the removal of anything `rm` can delete, regardless of any permission.

The `-r` option also needs to be used with care, as it stands for recursively delete. Ironically, the `-r` option takes a directory as a parameter, so it can recursively delete every file and directory under the passed location. Recursively deleting a file does not seem to make sense anyway. The following `rm` command shows an example to remove a file:

```
packt@PacktPublishing:~$ rm /tmp/testfile
```

Removing a directory

Removing a directory is done via the `rmdir` command; it, however, will only operate on an empty directory, as shown in the following command:

```
packt@PacktPublishing:~$ rmdir /tmp/testdir
```

Copying files and directories

To copy a file, the `cp` command can be used. Copying a file, you need to supply the source file and the destination file as parameters to `cp` in that order. Optionally, a directory can be supplied instead of a file to copy a directory. While `copy` takes many options, which the manual page explains in detail, the `-r` option can be important when dealing with directories, as it tells `copy` to recursively copy a directory and everything underneath it. The following command shows the use of the copy command:

```
packt@PacktPublishing:~$ cp /tmp/testfile
/tmp/testdir/copy_of_testfile
```

Moving files and directories

To move a file, the `mv` command can be used. Supply the source file and destination file as parameters to `mv` in that order. Optionally, a directory can be supplied for the source and the destination or just the destination.

Renaming a file is actually nothing more than moving a file from one name to another. The following command is used to move a file:

```
packt@PacktPublishing:~$ mv /tmp/testfile /tmp/testdir/moved_testfile
```

Changing file and directory access permissions

To grant or restrict access to certain files and directories, the `chmod` command can be used. This command stands for *change mode* and requires at least two parameters: the mode that needs to be applied and the file or directory on which this needs to be applied.

Managing permissions properly can be quite complex, though the manual page does help quite a bit. The basics are as follows. Under Linux, there are three standard access levels: user, group, and others. A fourth virtual-access level exists to cover the three others, all. Let us take a look at each of these in detail:

- **User**: This access level relates to the user who owns a file or directory; usually, it refers to the user who created the file or directory
- **Group**: This access level grants all the users who are also members of this group access to this level
- **Others**: This access level gives access to everybody else
- **All**: This is the fourth virtual-access level that incorporates the preceding three levels

The four access levels are often abbreviated with their first letters, `ugoa`. Next to the access levels, there are the access rights, and here, we will look at the three common ones. Technically, there are four, but more on that in a minute! The two primary access rights are read and write access to a file or directory, which are abbreviated with `r` and `w`. The third access right is execute, abbreviated with `x`, which grants execution permission on a file to a user, group, or anybody else. So, for example, `chmod` itself would require the execute access right to be set for anybody to actually execute that file. The fourth access right is `x` again, but this time it is applied to a directory. Since directories cannot be executed, the access right has a different meaning here and hence has four access rights. For directories, it allows users, groups, or anybody else to actually change into the directory and read the list of files in it.

Constructing a mode is done as follows. Firstly, the shorthand letter is used to designate the user, group, or others followed by `+` or `-` to grant or revoke access rights supplied immediately after. Users, groups, and other designators can be combined if separated by a comma. Refer to the following example to see the constructing mode:

```
packt@PacktPublishing:~$ chmod g+r-w,o+r-w-x /tmp/testfile
```

Very often, access permissions are applied by their numerical value, rather than through their letters. This has its roots mostly in history, where the actual mode bits were used. For more details on the numerical values, you can refer to the manual page.

Changing file and directory ownership

To change the owner of a file or directory, the `chown` command can be used, which stands for change owner. For this, two parameters are required: the new owner and the file or directory that requires new ownership, as shown here:

```
packt@PacktPublishing:~$ chown packt /tmp/testdir
```

To change the group membership of a file, a similar command to `chown`, called `chgroup`, which stands for change group, exists and works identically.

Changing passwords

To change passwords, the `passwd` command can be used. When executed without a parameter, the current user's password can be changed by supplying both the old and the new password. The root user can change any user's password by supplying that as the first parameter to `passwd`, as follows:

```
packt@PacktPublishing:~$ passwd
Changing password for packt.
(current) UNIX password:
Enter new UNIX password:
Retype new UNIX password:
```

Displaying the content of a text file

There are many tools to output the content of a file; `less`, `more`, or `cat`, to name just a few. They all work similarly, pass a filename as their parameter, and they will start displaying the content. Both `less` and `more` allow search or scrolling through the file, with `less` being more advanced than `more`. The `cat` utility, which stands for concatenate, will just output whatever it finds in the file, be it text or not, as shown here:

```
packt@PacktPublishing:~$ cat /tmp/testfile
```

One common operation used with `cat` is redirecting the output content of a file to somewhere else, be it a new file where its functions mimic the copying of a file or appending to another file.

There are a few programs that function in a manner that is very similar to `cat`, but operate on compressed files, decompressing them on the fly. Such commands are `zcat`, for **gzip** cat or `xzcat`, for the **xz** compression. A useful purpose lies herein that when redirecting the output, a file could be decompressed and the output can be written elsewhere. *Chapter 3*, *Installing an Operating System*, makes use of this by taking a compressed binary file and redirecting the output directly onto a flash disk, as shown here:

```
packt@PacktPublishing:~$ xzcat /tmp/archive.xz > /dev/sdb
```

Modifying the partitions on a disk

The `fdisk` command, which stands for fixed disk, is a command that can create and modify partitions on a hard or flash disk. It requires a disk device node to be supplied as a parameter. While it is quite menu-driven, `fdisk` has a lot of commands. The most important ones are briefly summarized, as follows:

- `m`: This command shows a help menu
- `p`: This command prints the current partition table
- `o`: This command wipes out the entire partition table and creates a new empty partition table
- `n`: This command creates a new partition by answering a few questions that `fdisk` asks
- `d`: This command deletes a partition
- `w`: This command writes the created partition table to the disk and exits

Always take great care when working with partitions.

The `fdisk` command does not actually write the changes to the disk unless explicitly requested. If there are errors, the *Ctrl* + *c* key can be used to quit `fdisk` without writing changes to the disk.

Formatting partitions

To format a partition, the various `mkfs` commands can be used. It depends on whether the supporting utilities are installed. When creating an `ext4` partition, `mkfs.ext4` is used. Likewise, to create a `fat` or `vfat` partition, `mkfs.vfat` is used. Each filesystem partitioning tool has different options and parameters, so the manual page for these commands should be checked for details. Generally speaking, when using the default settings, supplying the device-specific partition node, such as `/dev/sdb1`, for the first partition `(1)` on the second hard or flash disk `(b)` is passed as a parameter to the `mkfs` commands. Creating filesystems is a destructive operation. Use it with care!

In the following example, an `ext4` filesystem is created on a previously partitioned USB flash stick. Note that `sudo` was used here to obtain permission to write directly to the flash drive.

```
packt@PacktPublishing:~$ sudo mkfs.ext4 /dev/sdb1
```

A special variant of `mkfs` is `mkswap`, which creates a filesystem specifically geared to swap space.

Mounting partitions

Attaching storage to a system is called mounting. While many graphical desktop environments seem to just work, behind the scenes, they still mount and unmount disks and partitions. The `mount` command, which makes this happen, requires two parameters: the device node and the mount location.

Either of the two parameters might be omitted if either of them has been defined in the `fstab` file at `/etc/fstab`. The `fstab` file is parsed by `mount` to see what needs to be mounted, where, and how. Usage of the `mount` command is shown here:

```
packt@PacktPublishing:~$ mount /dev/sdb1 /tmp/testdir
```

Unmounting partitions

To remove a partition from a running system, the `umount` command is used and stands for *unmount*. The *n* seems to be thought about as being redundant so the term has been abbreviated to `umount`. It is very important to know that `umount` requires no files or directories that are being accessed or in use when detaching a partition from the system. Either the device node or the mount point can be used to unmount a partition, as shown here:

```
packt@PacktPublishing:~$ umount /dev/sdb1
```

Writing data

A somewhat unusual name is used for the program described in this section, `dd`. It is unknown what `dd` stands for, but its purpose is to copy data. There are many possible arguments to `dd`, but the most important ones used in this book will be covered here. The `if` parameter specifies the input file where data is to be read from. The `of` parameter is the output file parameter where the data is to be written. With these two parameters, it is already possible to copy data from the source to the destination. What makes `dd` so versatile is the plethora of other parameters. The `seek` parameter allows you to change the start position where to start writing. The `skip` parameter allows you to change the start position from where data is read. The `bs` parameter, which stands for block-size, determines the size of the data blocks involved in the transaction, and in combination with the `count` parameter, determines how much data is to be copied. As `dd` allows you to very specifically control a copy operation, it is often used to write full images, bootloaders at specific locations, and much more, as shown in the following command. See the manual page for more information.

```
packt@PacktPublishing:~$ dd if=inputfile of=outputfile seek=8 bs=1024
```

Changing to a special root directory

Normally, the root directory is the main system directory and everything branches from there. Sometimes, we want to restrict access to only certain parts of the system or temporarily pretend a certain directory is this root. The `chroot` command, which stands for change root, ensures that the supplied directory is considered the new root until exited. As a second parameter, `chroot` can be told what command to run from within this restricted root. In the following example, the root directory is changed to `/tmp/testdir` and the requested command to be executed, `bash`, will reside at `/tmp/testdir/bin/bash`, as shown here:

```
packt@PacktPublishing:~$ chroot /tmp/testdir /bin/bash
```

Forcing the system to write all content to disks

Modern systems buffer everything in memory and occasionally write that content to disk. The obvious reason for this is that the disks are very slow and the memory is fast. This does have a bad side effect, that is, sometimes the data that we expect to be on a disk is not actually written. The `sync` command causes all the data that is not yet written to disk to be synchronized to the disk, as follows:

```
packt@PacktPublishing:~$ sync
```

Adding new users

To add a new user to the system, the `useradd` command can be used. While there are many options and parameters that can be supplied, as seen in *Chapter 4*, *Manually Installing an Alternative Operating System*, the manual page does a great job of explaining all the options. However, just applying a new username is sufficient to create a bare user, as shown in the following command. Note that a new user does not have a password yet and needs one created via the previously mentioned `passwd` command.

```
packt@PacktPublishing:~$ useradd superpackt
```

Additional commands

This chapter contained a short list of the most basic commands. There are many more commands and even more guides on the Internet that go over a lot of commands. The website `http://www.reallylinux.com/` has a nice section called *Essential Commands* where these and more are briefly covered, but any site that covers the basic Linux commands can be used to learn more about commands.

Summary

This appendix covered the most basic Linux commands as well as the ones used throughout this book. The next appendix will give you an overview of the FEX configuration file.

C
The FEX Configuration File

Many systems have a way of configuring themselves, be it software or, in this case, hardware. Allwinner-based hardware is no different; there are some bits and pieces that do require configuring, such as GPIO pins. A chip cannot configure itself; that is, it cannot parse a configuration file and configure itself. The actual configuration of the chip is done by the various drivers. The procedure is illustrated in the following sections.

Initial boot up

The chip starts in a hardwired way, where certain components are preprogrammed to be active on specific pins. Because of this, the chip can boot from various boot media, as mentioned in *Chapter 3*, *Installing an Operating System*, and load the bootloader. The bootloader is also preconfigured to a certain hardware setup. Hence, as mentioned earlier, each board has its own bootloader.

Besides bringing up certain components, the bootloader has the following two important tasks:

- Loading the kernel into memory and later executing it
- Loading a configuration file into the memory for the kernel to use

However, the bootloader itself does not parse the configuration file.

It has to be mentioned that when using a mainline kernel, the principle is the same: the bootloader still loads a configuration file named device tree binary, which the kernel uses for configuration.

The FEX configuration concept is interesting in itself, where one file is changed to configure an entire device. As mentioned earlier, the FEX file is loaded into the memory by the bootloader, and the bootloader only checks one specific location. While u-boot is more flexible and could be configured to allow reading the configuration file from any location or any filename, the bootloader that is preprogrammed into the onboard NAND flash storage will only check the first partition on the device, and that has to be FAT formatted. As such, this is a common convention that we will follow for the remainder of this chapter. Additionally, this file has to be called by a specific name, `script.bin`, and the duplicate backup by the name `script0.bin` file. When booting a kernel that supports the onboard `nand` flash, the device node where this file will be stored is `/dev/nanda`. Otherwise, normal device nodes will be used, most commonly the SD card at `/dev/mmcblk0p1`.

Compiling and decompiling the FEX file

The `script.bin` file is, as its file extension suggests, a binary file. However, it is not possible to directly modify this file. The linux-sunxi community has created a set of tools to convert this binary file into a text file and vice versa. They can be found in their GitHub repository at `https://github.com/linux-sunxi/sunxi-tools`.

After cloning this repository, the `make fex2bin` command is run to build the **fexc**, the fex decompiler. Compiling and running this tool is probably best on a system that has a comfortable text editor available.

Running fexc to decompile the binary file will look like this:

```
[packt@packt:~]$ fexc -I bin -O fex script.bin script.fex
```

There are two shorthands in the form of symlinks to fexc, namely `fex2bin` and `bin2fex`. Using these makes the `-I` and `-O` parameters unnecessary.

Understanding the FEX file format

Using any text editor, a FEX file will show that it is divided in various sections, prepended by a header within brackets, `[]`. In the following example, the UART `0` component is explained:

```
[uart_para0]
uart_used = 1
uart_port = 0
uart_tx = port:PB22<2><1><default><default>
uart_rx = port:PB23<2><1><default><default>
```

Here, the component called `uart_para0`, the first serial port or UART, has four fields that a serial driver would read. Each field is set up as a key-value pair, where the key is on the left-hand side and the value is on the right-hand side of the equal sign. In this case, the key `uart_used` is set to the number `1` indicating that this definition should be parsed and activated.

Following that is the `uart_port` key, which is set to the number `0`, indicating that this configuration is about UART 0—the first UART port. Some of the settings are very straightforward key-value pairs, that is, a key on the left and a string or a number on the right. There is, however, a key-value pair that needs some special attention, and that is the pin configuration, where the value is a port definition. Every component might require certain pins to function.

A basic UART requires two pins: a transmit pin and a receive pin. The SoC can provide several UARTs on several pins. In the preceding example, two pins are defined, the transmit pin, `uart_tx`, and the receive pin, `uart_rx`. The pins on a SoC are very often grouped, usually by a related function.

In the case of the A10, it has nine groups called ports. Each port can consist of a varying amount of pins. Port B, for example, has 24 pins. The last two pins of the set are the UART transmit and UART receive pins. The numbering starts at 0, following from which, it should be no surprise that `PB22` and `PB23` in the preceding example are Port B: pin 22 and pin 23.

As mentioned before, each pin has multiple features, or as they say, many functions are multiplexed onto a pin. These multiplexes, or muxes, are enumerated, where MUX 0 always configures a pin as GPIO input, and MUX 1 always configures a pin as GPIO output. Depending on the port and pin, MUX 2 and beyond can have various meanings—in the case of `PB22` and `PB23` MUX 2 are the UART pins. The first parameter surrounded by angle brackets, `<>`, is thus defined as MUX 2.

Pin configurations

Chapter 8, *Blinking Lights and Sensing the World*, talks about the purpose of a pull-up resistor and a pull-down resistor. Allwinner-based SoCs actually have pull-up or pull-down resistors attached internally to the pins. The second angle-bracket surrounded parameter, `<1>`, in this case, enables the internal pull-up, a `<0>` here disables the pull-up/pull-down feature, and a `<2>` enables the pull-down feature. The pull-down feature is, however, only valid when the port is configured as an input; the SoC does not support pull-down on outputs. One more valid option is to use the keyword `<default>`, which tells the driver to use a safe default value.

The third angle-bracket surrounded parameter defines the current strength that the pin should output. There are four valid values: `0`, `1`, `2`, and `3`, where `0` corresponds to 10 mA, `1` to 20 mA, `2` to 30 mA, and `3` to 40 mA. The default value can be used to let the driver choose a safe default.

The fourth position defines the initial output level of the pin, which can be either low, `<0>`, or high, `<1>`. Naturally, this is only valid when configuring the pin as output. Also here, default means that the driver uses a safe default value.

While Port B: pins 22 and 23 were discussed and explained here, the rest of the pins and their muxes can be found on the linux-sunxi community wiki at `http://linux-sunxi.org/PIO`.

Further reading

The FEX file contains many other such components that can be set up, varying from configuring which pins are used for the SD card reader to what color format is used for the LCD. All options that have been discovered are also documented at the linux-sunxi wiki page, `http://linux-sunxi.org/Fex_Guide`. However, since each driver is written to read a key-value pair, things easily and often do change depending on the progression of the driver and kernel as a whole. When in doubt, the kernel source code can always be checked.

Installing the configured FEX file

Compiling the FEX file back into a bin file is nearly identical.

```
[packt@packt:~]$ fexc -I fex -O bin script.fex script.bin
```

Depending what boot medium is being used, `script.bin` has to be copied back so that the device can use these new, changed values. This can be the `/dev/nanda` partition on the onboard NAND flash or the first partition that is FAT formatted on a microSD card.

After having put the `script.bin` into place, a reboot is required for the system to read these changes.

Summary

This appendix gave a small introduction to the FEX file and showed how to modify it. The next appendix will try to cover the most basic troubleshooting for the most common pitfalls.

D
Troubleshooting the Common Pitfalls

When experimenting with new things, a lot can go wrong. In this appendix, a few of these common pitfalls are covered briefly. Anticipating everything that can go wrong is plainly impossible, but an overview of the most common problems is what this appendix is for.

Some general things that do tend to happen more often than one can imagine is the reading and entering of commands. A typo is easily made, or something is easily overlooked and read wrong. These things naturally happen and are all part of working with something exciting and new. So the first general tip is to always double-check your input. Despite the many eyes that went over all the pages in this book, there is always the possibility that a mistake has crawled into this book, so if something still goes wrong, even when following the book to the letter, check the errata.

Stability issues

Quite often, users find the many communities surrounding these devices and complain about strange crashes, unstable systems or random reboots. While it is, of course, always possible that the actual device might be damaged, very often a flaky power supply is to blame, which either provides unstable power or is not powerful enough.

Testing the strength or stability of a power supply is very difficult without additional equipment. What tends to happen if the power supply cannot deliver enough current is that the voltage starts to drop. This can be measured with a standard multimeter.

If the voltage drops below 4.8 volt, things are likely to go wrong. A very noisy power supply is even harder to test and requires an oscilloscope. It is probably best to get a well-known and good power supply. Phone chargers, for example, come in different strengths. There are cheap phone power supplies that barely deliver 500 milliampere, which, when the device is heavily stressed, is not enough. Obviously, there are also decent chargers that come with high-end smartphones that can easily supply 2000 milliampere, but even here, in combination with the hard disk or solid-state disk, the power requirement can be too high. It is thus advised to disconnect as many devices as possible. No USB device and no SATA storage as they get their power from the board. No other components that might receive power directly from the board. Ideally, only a power connection and a serial connection should be made, and a multimeter or another power measuring device should be used to see that the voltage does not drop too low.

Should the board still remain unstable even after all that, where other people with the same board using the same bootloader and ideally the same root filesystem have no problem, then the board might be defective. It is probably best to ask for an exchange where the board was purchased. But please, always try to verify everything else, as a board exchange is no fun for all the parties involved.

Boot failures when booting from SD cards

There is nothing more frustrating than spending hours on compiling and preparing an SD card to boot the board with, only to have it not work. While it is critical to have a serial console to see a debugging output during this stage, sometimes no output is generated at all due to an incorrect SD card or binary. The most important thing when experimenting with the bootloader is to always have a known working copy at hand. This is to make sure that modifications to u-boot are not the cause.

The nightlies from linux-sunxi are an alternative, as those should always work. See `http://dl.linux-sunxi.org/nightly/u-boot-sunxi/u-boot-sunxi/u-boot-sunxi-latest/` for a list of available bootloaders.

Sometimes, the microSD card might simply not be compatible or might even be broken. A different microSD card should be used to make sure the microSD card is not the cause. It is not unheard of that the first few bytes on the card are broken beyond repair. This is not generally a problem if the card is used as a simple storage medium. Because the bootloader gets written and read from the start of the SD card, this can thus result in a system that is unable to boot.

Now, going with either one of the nightlies or a self-compiled bootloader, quite often, users tend to choose the wrong file. This is not a big surprise, as there are many files and various instructions on the Internet that, sometimes, become outdated and are no longer accurate. A common example is that many older guides suggest writing `u-boot.bin` or `sunxi-spl.bin` using various `seek` offsets. Using the correct files and parameters, this can still work, whereas using the wrong parameters or wrong settings will cause failure. The recommended way, as described in *Chapter 4*, *Manually Installing an Alternative Operating System*, onto the board is to use `u-boot-sunxi-with-spl.bin` using a `blocksize (bs)` of `1024` and `seek` of `8`. Applying this to, for example, `sdb`, the following command should be used:

```
packt@packt:~# dd if=u-boot-sunxi-with-spl.bin of=/dev/sdb bs=1024
seek=8
```

This is because both the previous mentioned files are combined in one big file, so fewer things can go wrong. Finally, after writing the bootloader, the intention is to format the first partition to store the kernel. The partition number gets omitted and not `/dev/sdb1`, but `/dev/sdb` gets formatted. This is completely legal to do; having partitions on a storage medium is optional, though often useful. Formatting the entire disk, rather than the partition, however, has a side effect that the earlier written bootloader is now overwritten and gone. When the board boots, the bootloader is no more to be found and hence no output is generated.

No display output via a connected monitor

Quite often, when a monitor is connected via HDMI or VGA, there is no display output. This can be, understandably, very frustrating. Even more so, it probably worked fine with the preinstalled Android was working fine, but now when booting Linux from the SD card, nothing happens anymore. There are of course many possible reasons for not getting an image on the monitor, such as the cable might not be connected properly, the monitor might be set to the wrong input, and so on. Even though all these things seem very obvious and common, they do happen, and there are a few cases that are not so easily checked.

The Allwinner SoC is actually not very smart when it comes to displaying outputs. It needs to be told what is connected and what to use as an output. For this purpose, monitors and TVs, which really are just fancy monitors, have a communication channel in the cable called **Display Data Channel** (**DDC**). The purpose of the DDC is for the monitor to tell whoever asks what its capabilities and resolutions are. And this is where things get ugly.

For VGA, it was once agreed upon that there were two colors for the plugs: black if the port does not care about DDC and ignores it, and blue if it does read the DDC information and can respond accordingly. Unfortunately, many creators of development boards simply use a blue connector, thinking that is the standard VGA connector, but do not implement DDC. Without DDC, the board has no idea what is connected and how to send signals to it. And because of the ignorant usage of blue and black VGA plugs, there is no simple rule to follow anymore, which says that if your connector is black, you have to set up your VGA port manually.

For HDMI, the problem is very similar. The Allwinner HDMI display controller was created with television in mind for the most part. Thus, there are some quirks when connecting it to an HDMI monitor or when using a DVI to HDMI adapter. Android might, for example, have a purple hue over the image when being connected to a DVI monitor, which is a quirk caused by the driver in combination with the design of the display controller; running a linux-sunxi kernel usually fixes this.

The first thing that can be checked and changed is the `uEnv.txt` file that resides on the bootloader partition, possibly `/dev/sdb1`, when we use a microSD card in a USB adapter. Here, there should be a setting called `disp.screen0_output_mode=EDID:1280x720p60` or something similar for the bottom `extraargs` parameter.

This means that for the output, first try to probe the monitor via EDID, which is an extended form of DDC, and if that fails, fall back to the 1280 by 720 progressive resolution running at a refresh rate of 60 Hz. Some monitors might not properly announce their EDID information, so the first thing to try is removing the EDID bit from `output_mode` and thus forcing the output to `1280x720p60`. Additionally, it is possible that after EDID fails, the fixed resolution supplied is simply something the monitor does not accept. For valid resolutions and settings, check the second column in the table used in the following section. It should be noted that when we use a VGA monitor, EDID must be removed, as the driver will ignore the entire `output_mode` in that case. Additionally, the desired video output type needs to be appended to `extraargs` as `disp.screen0_output_type=4` for VGA or `3` for HDMI.

Overriding the kernel parameters via the `uEnv.txt` file might not be enough and cause things to not work. The same settings can and should also be supplied via the FEX file. The driver should, in theory, check both the kernel command line and the FEX file, but one might overrule the other, and thus, in case of trouble, both possibilities should be covered.

To force a video output mode, the FEX file needs to modified. The section that controls the display controller in the FEX file is the `[disp_init]` section. All the parameters for the display controller on the linux-sunxi wiki, `http://linux-sunxi.org/Fex_Guide`, there are a few things that need to be looked at. When getting things going for the first time, `disp_mode` is probably best set to `0` , indicating one frame buffer on `screen0`. Depending how the monitor is connected, `screen0_output_type` should be set to either `3` or `4`, where `3` is an HDMI display and `4` is a VGA display. The `screen1_output_type` object is best disabled by setting it to `0`. For `screen0_output_mode`, the following table should be used, and any setting applied to `screen1_output_mode` will be ignored:

`output_mode`	**Used for the TV/HDMI output**	**Used for the VGA output**
0	480i	1680 * 1050
1	576i	1440 * 900
2	480p	1360 * 768
3	576p	1280 * 1024
4	720p50	1024 * 768
5	720p60	800 * 600
6	1080i50	640 * 480
7	1080i60	
8	1080p24	
9	1080p50	
10	1080p60	1920 * 1080
11	pal	1280 * 720
14	ntsc	

The other settings should not be significant for the display output. The display should be able to output data now when booting, for example, the Fedora installation image, as was used in *Chapter 3*, *Installing an Operating System*.

Summary

This appendix went over the three most common issues, which most often happen when starting to work with the various development boards. The things covered in this appendix by no means cover every possibly scenario but should give you a decent strategy on tackling these early problems.

Index

D

F

M

N

O

P

R

S

T

U

V

W

X

Y

Thank you for buying
Getting Started with Cubieboard

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`.

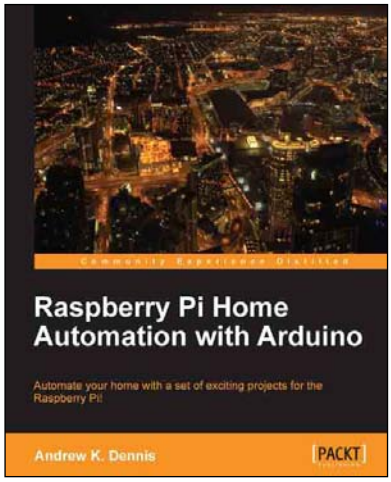
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

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.

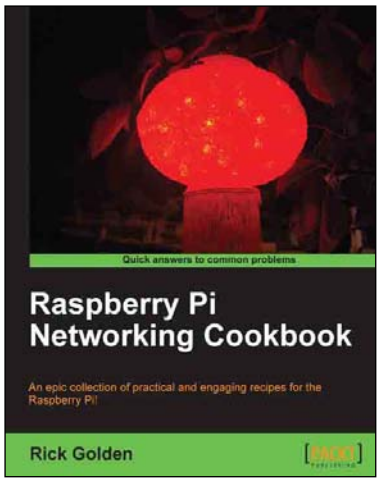


Raspberry Pi Home Automation with Arduino

ISBN: 978-1-84969-586-2 Paperback: 176 pages

Automate your home with a set of exciting projects for the Raspberry Pi!

1. Learn how to dynamically adjust your living environment with detailed step-by-step examples.
2. Discover how you can utilize the combined power of the Raspberry Pi and Arduino for your own projects.
3. Revolutionize the way you interact with your home on a daily basis.



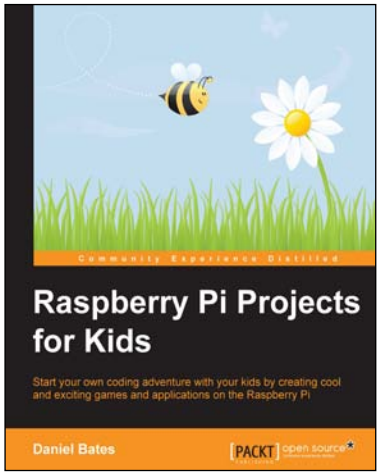
Raspberry Pi Networking Cookbook

ISBN: 978-1-84969-460-5 Paperback: 204 pages

An epic collection of practical and engaging recipes for the Raspberry Pi!

1. Learn how to install, administer, and maintain your Raspberry Pi.
2. Create a network file server for sharing documents, music, and videos.
3. Connect to your desktop remotely with minimum hassle.

[PACKT] open source
community experience distilled
PUBLISHING

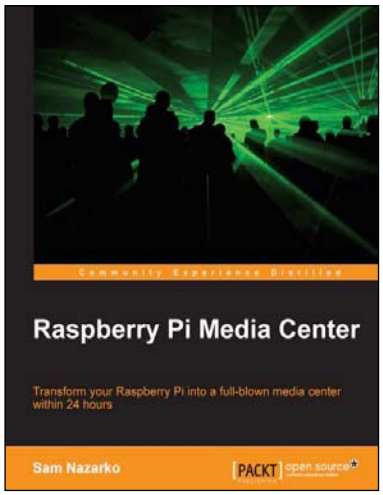


Raspberry Pi Projects for Kids

ISBN: 978-1-78398-222-6 Paperback: 96 pages

Start your own coding adventure with your kids by creating cool and exciting games and applications on the Raspberry Pi

1. Learn how to use your own Raspberry Pi device to create your own applications, including games, interactive maps, and animations.
2. Become a computer programmer by using the Scratch and Python languages to create all sorts of cool applications and games.
3. Get hands-on with electronic circuits to turn your Raspberry Pi into a nifty sensor.



Raspberry Pi Media Center

ISBN: 978-1-78216-302-2 Paperback: 108 pages

Transform your Raspberry Pi into a full-blown media center within 24 hours

1. Discover how you can stream video, music, and photos straight to your TV.
2. Play existing content from your computer or USB drive.
3. Watch and record TV via satellite, cable, or terrestrial.
4. Build your very own library that automatically includes detailed information and cover material.