Undergraduate Topics in Computer Science

Alan Holt
Chi-Yu Huang

Embedded Operating Systems

A Practical Approach

UTiCS

Springer

Undergraduate Topics in Computer Science

Undergraduate Topics in Computer Science (UTiCS) delivers high-quality instructional content for undergraduates studying in all areas of computing and information science. From core foundational and theoretical material to final-year topics and applications, UTiCS books take a fresh, concise, and modern approach and are ideal for self-study or for a one- or two-semester course. The texts are all authored by established experts in their fields, reviewed by an international advisory board, and contain numerous examples and problems. Many include fully worked solutions.

More information about this series at http://www.springer.com/series/7592

Alan Holt · Chi-Yu Huang

Embedded Operating Systems

A Practical Approach

Springer

Alan Holt
IP Performance
Bristol
UK

Chi-Yu Huang
Tata Technologies Ltd.
Bristol
UK

ISSN 1863-7310 ISSN 2197-1781 (electronic)
ISBN 978-1-4471-6602-3 ISBN 978-1-4471-6603-0 (eBook)
DOI 10.1007/978-1-4471-6603-0

Library of Congress Control Number: 2014948743

Springer London Heidelberg New York Dordrecht

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

In memory of Marjorie Rose Holt
and Siou-yìn Zheng Huang

Preface

Most people are aware of mainstream operating systems such as Microsoft's Windows or Mac OS X. This is because these are the operating systems that they directly interact with on their personal computers. However, personal computers rarely operate in isolation nowadays.

Many computing applications require *online* access. Access to the Internet relies on many computing systems, for example, Ethernet switches, packet routers, Wireless access-points and even DNS servers. Unlike personal computers, network infrastructure equipment perform dedicated tasks and are often described as embedded systems.

The term embedded system covers a range of computing systems. It derives from computing systems that are *embedded* within some larger device, for example, within a car's engine management system. The term has clearly broadened to cover standalone systems too. Nor is it limited to network infrastructure systems or consumer goods such as satellite navigation devices.

The term embedded system is applied to devices which are small and have limited resources (relative to a personal computer). Mobile devices such as smart phones and tablets are, therefore, described as embedded devices, even though their function closely resembles a personal computer. Nevertheless, an embedded system needs an operating system specific to its needs, for example:

- Small form factor.
- Low processing power and small memory.
- Reduced power consumption for battery longevity.
- Support for real-time applications.

There are many embedded operating systems available, however, in this book we confine our discussion to the GNU/Linux operating system. GNU/Linux is not exclusively an embedded operating system and is used on personal computers and servers as Windows and OS X. However, due to its open source nature it is highly customisable and can be adapted to many environments. For this reason, GNU/Linux systems have gained considerable popularity in the embedded system market.

This is not to say that GNU/Linux is a panacea for embedded systems and there are many excellent alternatives. Nevertheless, there are some good reasons for choosing GNU/Linux:

- GNU/Linux is open source and is freely available.
- Due to its open source nature, GNU/Linux is highly customisable so we can build bespoke systems specific to our needs.
- It is widely used for embedded systems.
- GNU/Linux, like its Unix predecessor, was used extensively in education to teach operating systems.
- While the subject of the book is embedded operating systems, our choice of GNU/Linux means it could be used as a text for a more general course on operating systems.

Operating systems is a diverse subject area and there many books on the subject and many on GNU/Linux alone, covering topics such as the kernel, administration, networking, wireless, high-performance computing and systems programming (to name a few). There are even books on GNU/Linux embedded systems.

However, we felt there was a gap in the literature which described the component parts of an operating system and how they worked together. We address these issues in this book by adopting a practical approach. The procedure for building each component of the operating system, namely the bootloader, kernel, filesystem, shared libraries, start-up scripts, configuration files and system utilities, from its source code, is described in detail. By the end of this book the reader will be able to build a fully functional GNU/Linux embedded operating system.

We take this opportunity to pre-empt any Amazon reviews stating this book is not for beginners. Indeed, this is not an introductory text on operating systems, rather it is aimed at undergraduate/graduate level students and industry professionals.

Acknowledgments

The authors would like to thank the following people for the valuable contribution they made to this book: Paul While and Edwin Chen.

Contents

1 Introduction 1
- 1.1 An Overview of Operating Systems 1
- 1.2 Overview of Embedded Systems 3
- 1.3 Brief History of Linux and GNU 5
- 1.4 GNU/Linux as an Embedded Operating System 6
- 1.5 Virtualisation 7
- 1.6 Conventions Used in this Book 9
- 1.7 Book Outline 10

References 11

2 Overview of GNU/Linux 13
- 2.1 The Bootloader 14
- 2.2 The Kernel 15
- 2.3 The Init Process 16
- 2.4 The Root Filesystem 17
- 2.5 Process Management 20
  - 2.5.1 Signals 20
  - 2.5.2 Job Control 23
- 2.6 Process Input/Output 24
- 2.7 The Process Environment 28
  - 2.7.1 Shell Parameters 32
  - 2.7.2 Quoting 33
  - 2.7.3 Positional Parameters 34
  - 2.7.4 Special Parameters 35
- 2.8 Summary 37

References 37

3 The Filesystem in Detail 39
- 3.1 GNU/Linux File Space 40
  - 3.1.1 Permissions 42
- 3.2 The Filesystem 53
  - 3.2.1 Pseudo Filesystems 57

3.3 Partitions . . . 59
3.3.1 Partitions and Boot Sectors . . . 59
3.3.2 Extended Partitions . . . 66
3.4 Summary . . . 69
Reference . . . 69

4 Building an Embedded System (First Pass) . . . 71
4.1 Creating the Root Filesystem . . . 72
4.2 Building a Linux Kernel . . . 76
4.3 Build the Root Filesystem . . . 78
4.4 Running UML . . . 79
4.5 Networking . . . 81
4.6 Summary . . . 84
Reference . . . 85

5 Building an Embedded System (Second Pass) . . . 87
5.1 Preliminaries . . . 89
5.2 Glibc . . . 89
5.3 Optimisation . . . 92
5.4 Ncurses . . . 92
5.5 Busybox . . . 93
5.6 Bash . . . 94
5.7 Sysvinit . . . 95
5.8 Devices (/dev) . . . 96
5.9 Administrative Files and Directories . . . 97
5.10 Start-up Scripts . . . 102
5.11 Test with UML . . . 104
5.12 Running the System Natively . . . 105
5.12.1 Compiling the Kernel . . . 106
5.12.2 The Filesystem . . . 106
5.12.3 The Bootloader . . . 110
5.12.4 Booting the System . . . 111
5.13 Summary . . . 113
References . . . 114

6 Compiler Toolchains . . . 115
6.1 GCC . . . 116
6.1.1 The Preprocessor . . . 117
6.1.2 Optimisation . . . 119
6.1.3 The Assembler . . . 121
6.1.4 The Linker . . . 122
6.2 Build an ARM Cross Toolchain . . . 128

6.3 Binutils . . . 130
6.3.1 GCC (Bootstrap Compiler) . . . 131
6.3.2 Newlib . . . 132
6.3.3 GCC . . . 133
6.4 Testing the Toolchain . . . 134
6.5 Summary . . . 136

7 Embedded ARM Devices . . . 137
7.1 Raspberry Pi . . . 138
7.1.1 Installing an Operating System . . . 139
7.1.2 Using the Raspberry Pi Serial Port . . . 140
7.1.3 Remote Serial Port . . . 144
7.2 BeagleBone . . . 151
7.2.1 Setting Up the BeagleBone . . . 152
7.2.2 Physical Computer Programming on the BeagleBone . . . 156
7.3 Summary . . . 160
References . . . 160

8 OpenWRT . . . 161
8.1 Open-Mesh . . . 162
8.1.1 Building OpenWRT . . . 163
8.2 Dragino . . . 167
8.2.1 Booting up the Dragino for the First Time . . . 168
8.2.2 Running Arduino Yùn Firmware . . . 171
8.3 Programming the M32 Unit . . . 178
8.4 Summary . . . 181
References . . . 181

Appendix A: Start-up Scripts . . . 183

Appendix B: Inittab . . . 191

Index . . . 193

Abbreviations

API	Application programming interface
Bash	Bourne again shell—a command-line interpreter
Busybox	A collection of Unix utilities designed for embedded systems
Debian	A GNU/Linux distribution
GCC	GNU compiler collection
Glibc	GNU C Library
GNU	GNU is not Unix
Grub	Grand unified bootloader
Ncurses	A library of screen handling functions
NSS	Name switch service
TCP/IP	Transmission control protocol/internet protocol
Tmpfs	Temporary file system
Ubuntu	A GNU/Linux distribution based upon Debian
UML	User mode Linux
Vi	A text editor
VM	Virtual machine

Introduction 1

Most people are familiar with general purpose computing devices, such as desktops and laptops. Their use is common-place and support a wide variety of applications, many of which involve a wider access to distributed applications over the Internet (electronic mail, social media etc). Users interact with general purpose computers directly through keyboards, mice and monitor screens. There are many consumer devices, such as mobile phones, tablet computers and satellite navigation devices, that are classified as embedded devices. They support user interaction through touch screens, microphones, audio speakers and accelerometers.

Nevertheless, many embedded systems operate in the background with little or no direct human interaction. Embedded computer systems are used extensively throughout our technological society.

As we look to the future we should contemplate an *Internet-of-things* (IoT). IoT predicts the attachment of computing devices (with network capabilities) to *objects*. The attributes of such an object can be monitored and digitised by its associated computing device which then relayed (over a network) to remote system for analysis.

A precursor to the Internet-of-things is smart energy metering like that of 3E-Houses [1]. Energy monitoring devices were installed in social housing to record energy consumption in domestic homes. The data was transmitted over the Internet to central database. This data was analysed and fed back to the participants of the project to increase their awareness of their energy usage. The hope was, that through this feedback mechanism, people would reduce their energy consumption and CO2 footprint.

1.1 An Overview of Operating Systems

Operating systems are software components dedicated to the management of the computer system's hardware. Most computer systems (embedded or otherwise) operate under the control of an operating system. Unix-like operating systems, such as GNU/Linux, comprise a kernel, software libraries and a number of utility programs.

A. Holt and C.-Y. Huang, *Embedded Operating Systems*, Undergraduate Topics in Computer Science, DOI 10.1007/978-1-4471-6603-0_1

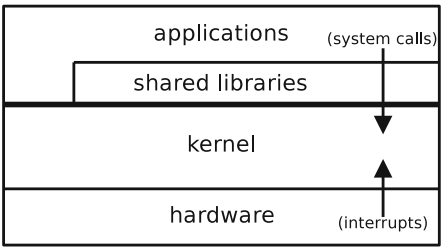


Fig. 1.1 Operating system overview

The diagram in Fig 1.1 shows the operating system architecture. A piece of software called a bootloader is also required to start the kernel when the computer system is powered. We discuss the bootloader in more detail in Chap. 2.

The kernel is a program that manages the resources of a computer system, It allocates these resources between separate process in a controlled way. Processes are computer programs under execution and may be initiated by different users. Resources are, therefore, not just allocated amongst processes, but also amongst users. What distinguishes the kernel from the rest of the operating system software, is the processor privilege level the respective code is executed. Most modern processors support multiple privilege levels. Intel 386 processors, for example, support four privilege levels. GNU/Linux only uses two, level 0 (the most privileged) for the executing kernel code and level 3 (least privileged) for running user code.

The reason for these different privilege levels is to prevent user code from directly accessing certain hardware resources. For example, I/O port instructions can only be accessed by kernel code. Clearly, user processes require access of computer systems peripheral devices (I/O port instructions). A process can context switch into kernel mode by issuing a software trap. In kernel mode, a process can access hardware via the kernel's own subsystems. Access to hardware is, therefore, controlled by the kernel. Functions of the kernel can be characterised below:

Process management Computer systems store a number of programs in their memory. These programs may be part of the operating system's utilities or software written by a user as an application. A program under execution is called a *process*. The system may execute multiple processes concurrently. The process management system gives the illusion of processes running simultaneously, typically by giving each process a *time slice* of the CPU (or CPUs). A scheduler makes the decision as to which process is allocated a time slice. A process may have its execution suspended and resumed several times before it terminates.

Memory management Memory management is a complex function. In order to manage memory efficiently and effectively, the kernel implements *virtual memory*. Virtual memory gives appearance that a process has more memory than it actually has (or even, more than the entire system has). The virtual and physical memory are divided into *pages*. With paged memory, a virtual address is divided into two parts, namely, a virtual page frame number plus an offset into the page. When a virtual memory location is referenced, the virtual page frame is translated into

a physical page frame number. The offset is then used to identify the specific memory location within the page.
Filesystem Persistent data is stored in files which in turn, are organised in directories. The filesystem provides the functions and data structures required to manage files and directories (as well as other files objects) within a disk partition.
Device management In addition to the CPU and main memory, a computer system will comprise a variety of peripheral devices. For example, disk drives, network interfaces, input/output devices etc. Device drivers are pieces of software within the kernel that control the devices.

1.2 Overview of Embedded Systems

Embedded systems are hard to define. The problem lies in the fact there are exceptions to any rules one may conceive of. While it is not easy to find a characteristic that is exclusive to embedded systems we examine a few below:

A Subsystem of a device or machine Such systems are *embedded* within a more sophisticated device or machine, an engine management system of a car or microprocessor within an appliance, for example. Given this definition, one may view a general processing computer as being made up of a number of embedded systems. In addition to the CPU and main memory, general processing computers also include bus controllers, disk controllers, network interfaces and video controllers. All of which could be considered embedded systems in their own right. Furthermore, there are numerous *non-subsystem* examples of embedded systems, WiFi access-points, network routers, set-top boxes, for example.
Dedicated application Embedded system are largely designed to perform a specific task such as monitoring a temperature sensor or controlling a valve. In the case of consumer electronics, however, devices perform multiple tasks. Modern mobile phones are rarely limited to just making telephone calls. Many are described as *smart* phones and can support a plethora of applications. Given the increase in the power and sophistication of embedded devices it is difficult to distinguish them from general processing systems.
Small footprint A small footprint is a typical characteristic of embedded systems. This is usually driven by the task it has to perform.
Low power consumption Many embedded systems are battery powered and therefore need to conserve energy consumption. Such systems may be static or mobile, either way they operate at a distance from any mains power. There are obvious exemptions to this rule. High speed routers and switches, for example, have powerful processors for forwarding high volumes of network packets. Consequently energy consumption will be high and mains supplied.
Low processing power and memory This is somewhat a corollary of the characteristic above. Whereas, in the past embedded system were almost exclusively based

on specially designed hardware, it is possible nowadays to develop embedded systems on commercial off-the-shelf (COTS) hardware.

Real-Time systems Real-time systems are systems that must respond to events within strict time constraints. The term "real-time system" is sometimes used synonymously with embedded systems. Real-time systems are almost always dedicated to a single application and can, therefore, be categorised as embedded systems. However, not all embedded systems support real-time applications.

Requires specially designed operating system This characteristic may have been true in the past when embedded system had limited processing and memory resources. It is still true to some extent but the increase in processor speed and memory size means that operating systems that run on general purpose computing platforms can also run used to control embedded systems. The Linux kernel for an embedded system will almost certainly need to be customised to work on the target platform (but this is true of any computer system). For devices that lack memory management units (MMUs) there is separate Linux kernel branch called uCliinux. Systems that need to support real-time applications need a real-time kernel. The vanilla Linux kernel does not support real-time. There is a real-time version Linux, namely, RTLinux which has an RTOS (real-time operating system) microkernel.

Interact with the physical world Physical computing is a branch of computing concerned with sensing and responding to the analogue world. This application is considered domain of embedded systems.

Single board computer Entire computer systems can be built on a single printed circuit board (PCB). The PCB includes the CPU memory and peripheral devices. Such systems are called single board computers (SBCs) and are well suited to embedded system platforms. It is possible to go beyond single *board* computers with system on a chip (SoC). This format too, lends itself to embedded system design.

The distinction between an embedded system and a general processing system is somewhat blurred. The Raspberry Pi for example, fits many of the criteria above, yet is primarily designed to function as a general processing computer. However, given that it shares many characteristics of an embedded system, it is being extensively for embedded applications.

We do not intend to get overly concerned with strict definitions of embedded systems in this book. Our focus is the core components of an embedded operating system, therefore, we neglect graphical user interfaces and sophisticated user input devices. Nor are we concerned with the processing of multimedia such as audio and video. We accept that consumer electronics systems, which are considered to be embedded systems, do support sophistacted GUIs (touch screens, etc) and multimedia. However, these components are outside the scope of this book.

We complete this section with a list of (typical) embedded system applications. This is by no means an exhaustive list but gives some idea of the scope of the subject area:

- Consumer electronics: cameras, mobile phones portable gaming consoles, Satellite navigation and GPS handsets.
- Automotive electronics: Cruise control and navigation.
- Communications equipment: base stations, network routers and wireless access-points.
- Factory automation equipment: robots, sensor and inventory control systems.
- Office automation equipment: Scanners, photocopiers and printers.
- Home automation: Security systems, Temperature control Smart meters.

1.3 Brief History of Linux and GNU

Linux is a *Unix-like* operating system developed by Linus Torvalds. To understand the origins of Linux, we present a brief history of the Unix operating system and some of its derivatives.

Unix was developed in 1969 by Ken Thompson, Dennis Ritchie Douglas McIlroy AT&T Bell Laboratories. Unix is a multi-user, time-sharing operating system. It was first developed for medium-sized computers, but has been ported to personal computers and super computers. The reason that Unix was so portable, was that it was developed in a machine independent language. Typically, the development of an operating system was tightly coupled to the computer system it was designed for. This was because operating systems were written in machine dependent languages (assembly language of machine code). While the first versions of Unix were developed in assembler, later versions were developed in the high-level language C [2] (also conceived by the Bell Laboratories). Unix System III was released in 1983, followed by Unix System V in 1983 which AT&T was able to market after its divestiture.

BSD Unix was developed by the Computer Systems Research Group (CSRG) at the University of California, Berkeley. The initial BSD Unix was an augmentation of the Bell Labs Unix. There have been a number of Unix-derivatives based on BSD. For example, SunOS from Sun Microsystems is a version of BSD Unix (one of the co-founders of Sun, Bill Joy was a BSD developer). Later, SunOS was re-branded as Solaris and was based on System V Release 4 (see below).

As Unix matured, CSRG no longer considered BSD Unix a research interest and terminated its development. A number of Unix-like, open-source operating systems descended from BSD, namely, FreeBSD [3,4], NetBSD, and OpenBSD [5]. Mac OS X is based on the Mach kernel and is derived from the BSD implementation of Unix in Nextstep. Nextstep was the object-oriented operating system developed by Steve Jobs when he was at NeXT after leaving Apple in 1985.

In 1987, AT&T (in collaboration with Sun Microsystems) merged System V release 3, BSD (Berkeley Software Distribution) and Xenix to form System V Release 4 (SVR4).

Attempts to standardise Unix in the late 80s and early 90s gave rise to the Unix wars. This adversely affected its acceptance in the market. The alliance between

AT&T and Sun Microsystems concerned many other Unix vendors. This gave rise to two opposing factions, the X/Open consortium and OSF (Open System Foundation).

In 1991, AT&T "spun off" its Unix System Laboratories (USL) and Unix when it sold the Unix operating system. The following year, the Novell Corporation signed a letter of intent to purchase USL and Unix. The transaction was completed in 1993. The same year, Novell sold its entire Unix business to the Santa Cruz Operation (SCO).

The GNU (GNU is Not Unix) project was initiated by Richard Stallman to produce a free Unix-like operating system. The intended kernel for the GNU system was the Hurd kernel. Hurd stands for "HIRD of Unix-Replacing Daemon" and HIRD stands for HURD of Interfaces Representing Depth. Despite the wealth on recursive (and circular) acronyms, HURD never became the GNU kernel due to its slow development. The BSD kernel was considered, but was ruled out due to the AT&T lawsuit against BDSI.

Minix is a Unix-like operating system developed by Andrew S. Tanenbaum. Tanenbaum developed Minux for educational purposes. Released under the BSD license, Minix is free and open source software. It was Minix that largely inspired Linus Torvalds to develop the Linux kernel. Linux was initially released under Torvalds' own license but later it was released under GNU General Public License. Making the Linux source code available to the wider community meant others could contribute to its development. Development of the GNU software was also required to integrate it with the Linux Kernel. GNU/Linux operating systems are disseminated as *distributions*.

1.4 GNU/Linux as an Embedded Operating System

GNU/Linux was not developed specifically for the embedded system market. However, given that it can easily be customised, it is well suited to embedded projects. There are a number of advantages to using GNU/Linux as an operating system for embedded systems:

- GNU/Linux is open source, therefore, there are no licence fees.
- The software is fairly mature and, therefore, stable.
- There is a large community of developers.
- Access to the source code means that modifications can be made to suit the application.

There are some disadvantages of course. The Linux kernel was designed for multiuser, timesharing systems. It is, therefore, quite large and complex. For some applications GNU/Linux may be considered overly complex. The kernel also lacks support for real-time which is a requirement of some embedded applications. There are a couple of solutions to this limitation. There are a number of patches (such as PREEMPT-RT) which enable real-time support within the Linux kernel. Also, there

is RTLinux which is an RTOS[1] microkernel that runs the Linux kernel as a preemptive process.

The current Linux kernel is designed to work with a memory management unit (MMU) which many embedded platforms lack. μCLinux is a Linux kernel fork designed to work with processors which do not have an MMU. There are a number of GNU/Linux distributions specifically designed for embedded systems, a few a listed below:

OpenWRT Linksys used firmware for the WRT54G series of routers based upon code that was licensed under GPL (GNU public license). Linksys, therefore, had to release their modified code enabling community developers to produce new versions of the firmware. The OpenWRT project originated from the initial development. OpenWRT support a wide variety of process architectures. It uses the ipkg package management system. We discuss OpenWRT in more detail in Chap. 8.
Arch Linux The Arch Linux distribution is primarily for i686 and x86-64 architectures but there is also a distribution fo ARM processors. It uses the Pacman for package management, which was developed specifically for Arch Linux.
Angstrom Angstrom distribution supports a variety of embedded devices. The distribution derives from a number of projects, namely, OpenZaurus, OpenEmbedded, and OpenSIMpad. Angstrom is used on the BeagleBone and BeagleBoard platforms. The package management system for Angstorm is opkg.
Android Android is designed for tablets and smart phones It was developed by Google and the Open Handset Alliance.

1.5 Virtualisation

Operating systems provide a layer of abstraction above the computer system's hardware. The operating system virtualises many of the physical resources of a computer.

- Memory
- Disks and disk partitions
- Network interfaces and bridges

Modern computers are sufficiently powerful to use virtualization to present the illusion of many smaller virtual machines (VMs), each running a separate operating system instance. This has led to a resurgence in the interest of VM technology Virtualisation is not a new concept. Virtual memory was conceived in the 1950s. IBM introduced the VM/370 operating system for its 370 architecture mainframes in the 1960s. VM/370 environment comprises two software components: the control program (CP) and the conversational monitor system (CMS). CP performs resource

[1] RTOS is a real-time operating system.

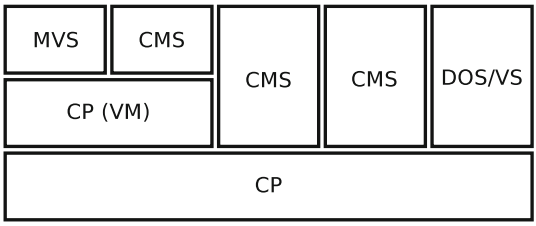


Fig. 1.2 VM/370 Architecture

management and creates the virtual machine environment in which an operating system can run.

CMS is an interactive development environment. It runs as a guest in a CP virtual machine (VM). CMS provides program development and personal computing functions in an interactive fashion to an individual terminal user. That is, CMS gives each user the illusion they are the sole user of a VM/370 operating system, While each CMS user uses a separate VM, the code is shared between them. Other operating systems, such as DOS/VS, MVS, OS/MFT, OS/MVT and SVS can also run under CP. CP can even run in a VM under the control of CP itself.

Guest operating systems run under the control of a hypervisor. There are two classifications of hypervisors, namely Type 1 and Type 2. Type 1 hypervisors run natively on the host's hardware. Type 2 hypervisors are *hosted*, that is, they run on a host operating system. Below we discuss a number of virtualisation technologies:

Quick emulator (Qemu) Qemu is a hosted hypervisor virtualisation technology. Qemu supports number of operating systems, including GNU/Linux, BSD, Solaris and Microsoft Windows. It also support a number of architectures, namely, x86, ARM, MIPS, PowerPC and SPARC (amongst others).
Bochs Bochs is an emulator for PC compatible hardware. It is distributed under the GNU Lesser General Public License (GLPL).
VMware VMware is a company that specialises in virtualisation technology. They offer both type 1 (ESX Server) and type 2 (VMware Workstation and VMware Player) hypervisor products.
XEN VMM XEN VMM (virtual machine monitor) is a type 1 hypervisor (that is, it runs directly on the hardware). Each guest runs as a *domian*. Dom0 is a special domain which controls the hypervisor. Dom0 is alos used to start and stop VMs (domUs). XEN supports two virtualisation methods, hardware virtual machine (HVM) and para-virtualisation. Para-virtualised guests require a modified version of the kernel, whereas HVM guests do not.
User Mode Linux (UML) UML is a modification of the Linux kernel so that it can run as user process on a (regular) host kernel. No hardware virtualisation is done with UML. We use UML in this book as a tool for testing embedded operating systems. Figure 1.4 shows the UML architecture.

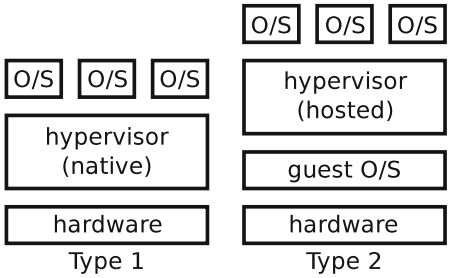


Fig. 1.3 Type 1 and Type 2 Hypervisors

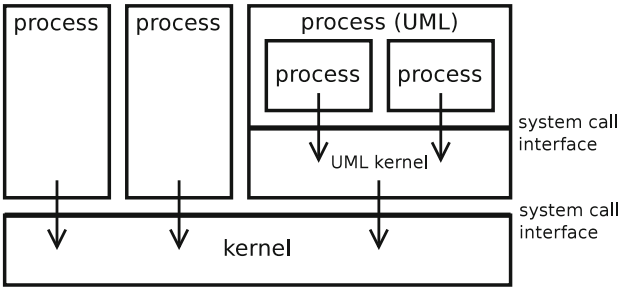


Fig. 1.4 User mode Linux

1.6 Conventions Used in this Book

We use a GNU/Linux platform throughout this book. Note, we use the term "GNU/Linux" rather than the more conventional (but erroneous term) Linux only refers to the kernel. The rest of the operating software (or most of it at least) is GNU. Despite this, Linux is almost exclusively used to reference the whole operating system. Nevertheless, we use the term GNU/Linux in recognition of both the kernel developers and the developers of other essential components of the operating system. Sometimes we use the term "Unix" when referring to a feature that is common to all Unix and Unix-like operating systems such as GNU/Linux. We use the term "Linux" to refer to the enitre operating system (rather than just the kernel) only when it is used that way by others. "Linux".

Most of the examples are performed on a general purpose Intel 386 based platform running Debian Squeeze. We recommend you replicate this environment used in this book. However, we would expect most of the examples to work under most Debian derivatives, such as Ubuntu and Mint. If you cannot allocate a dedicated processor to running Debian Squeeze, then you could use the debootstrap utility to create a Squeeze build environment. You can then chroot to the system's root directory. This is indeed what we did for the purpose of writing this book.

Some examples are performed on actual embedded GNU/Linux systems. Often, this necessitates running commands on both the embedded system and the general purpose machine. The text explains which commands are run on which platform. However, in order to make it clear, we use different command-line prompts to distinguish between them. A single dollar sign $ is used to denote the general purpose processor. The $ also means that commands are run as a regular user. If the prompt is a # then commands are run as superuser. Commands are seldom run while logged in to the superuser account, instead we use the sudo utility when we need to run commands with superuser privileges.

Most of the examples given in this book are from the GNU/Linux command-line. Throughout we assume Bash as the default shell. Input to and Output from the command-line is in a monospaced font. The example below shows the date command entered at the command-line prompt ($). Below this line is the output of the command:

```
$ date
Mon Sep  9 14:36:51 BST 2013
```

Names of files (including absolute and relative pathnames) are highlighted in italic, */etc/hostname*, for example.

1.7 Book Outline

This book is organised as follows:

Chapter 2 Overview of GNU/Linux. In this chapter will give a brief overview of the GNU/Linux system. We introduce the components of the system, such as the bootloader, kernel and filesystem. We also cover process management, process input/output and the process environment.

Chapter 3 The Filesystem. Access to persistent storage is supported through the *filesystem*. ‘ GNU/Linux support many filesystem types by using a virtual filesystem (VFS). In this chapter we discuss GNU/Linux filesystems in detail.

Chapter 4 Building an Embedded System (first pass). In this chapter we describe how to build an embedded systems using the debootstrap utility. We build the system to run under a virtual machine. The virtualisation technology we use is user mode Linux (UML).

Chapter 5 Building an Embedded System (second pass). We build an embedded system to run natively on an actual processor. We build all the system components from the source code, including the kernel, bootlader, shared libraries and Unix utilities. We also create the boot scripts and system configuration files.

Chapter 6 Compiler Toolchains. In this chapter we introduce the concept of the compiler toolchain (or just toolchain). Compilers translate instructions from a high-

level programming into machine dependent binary executable code. The "compiler" is rarely a single program, but rather a suite of programs, hence a *toolchain*. We also describe how to build a cross toolchain so that programs for a specific target architecture can be built on a host of a different architecture. This is particularly useful for building embedded applications because on embedded systems, compiling source code can be resource intensive. Embedded devices are typically low in CPU and memory and, therefore, running compiler toolchains on the actual target can be impractical.

Chapter 7 In this chapter we look at two ARM based platforms: BeagleBone and Raspberry Pi. The BeagleBone and BeagleBone Black are part of the BeagleBoard series single-board computers. produced by Texas Instruments and Digi-Key. They are aimed at the educational and hobbiest market.

The BeagleBones are shipped with the Angstrom GNU/Linux distribution. The Raspberry Pi was designed as inexpensive PC and aimed at the educational market. However, its small form factor makes it ideal for embedded system use. The Raspberry Pi can run a number of GNU/Linux distributions as well as a few non-GNU/Linux operating systems.

Chapter 8 OpenWRT is a GNU/Linux distribution for embedded systems. OpenWRT originally came out of development of the firmware for the Linksys WRT54G home router series. The project, however, quickly expanded to many other systems. In this chapter we focus of two particular devices that run the OpenWRT operating system, namely the OMxP Open-mesh devices and the Dragino MS14 series. We show how build OpenWRT firmware images for both of these units.

References

1. Holt A (2013) Get smart. Linux Mag 5:47–51
2. Kernighan BW, Ritchie DM (1978) The C programming language, Prentice-Hall software series, Prentice-Hall, Englewood Cliffs
3. Lehey G (2003) The complete FreeBSD. O'Reilly, USA
4. McKusick MK, Neville-Neil GV (2004) The design and implementation of the freeBSD operating system. Pearson Education, Upper Saddle River
5. Lucas M (2003) Absolute openBSD: unix for the practical paranoid. No starch press series. No Starch Press, San Franscisco
6. Maurice J (1986) Bach: the design of the UNIX operating system. Prentice-Hall Inc, Upper Saddle River
7. Barrett DJ, Silverman RE (2001) SSH, the secure shell. O'Reilly, USA
8. Benvenuti C (2006) Understanding linux network internals. O'Reilly, USA
9. Chastain ME (1999) Ioctl numbers. Linux kernel source, filename: documentation/ioctl-numbers.txt, 10
10. Corbet J, Rubini A, Kroah-Hartman G (2005) Linux device drivers, 3rd edn. O'Reilly Media Inc, USA
11. Haviland K, Salama B (1987) Unix system programming. Addison-Weseley, Boston
12. Kroah-Hartman G (2009) Linux kernel in a nutshell. O'Reilly Media inc, USA

13. Rubini A, Corbet J (2001) Linux device drivers. O'Reilly, CA
14. Stevens WR (1994) TCP/IP illustrated: the protocols, vol 1. Addison-Weseley, Boston
15. Stevens WR, Fenner B, Rudoff AM (2003) UNIX network programming, vol. 1, 3 edn. Pearson Education, Upper Saddle River
16. Venkateswaran S (2008) Essential linux device drivers, 1st edn. Prentice Hall Press, Upper Saddle River

Overview of GNU/Linux

2

The GNU/Linux operating system software comprises four components:

- Bootloader.
- Kernel.
- Shared libraries.
- System commands and utilities.

A full GNU/Linux distribution will also comprises a large collection of user application software, such as email clients, web browsers, database management systems etc.

The bootloader is the first piece of software to execute and is primarily responsible for loading the kernel. Due to space constraints, the bootlader may be split into stages. The first stage, which typically, very small, is located at a specific area of system's disk (or other non-volatile memory). The first stage bootloader loads the next stage of the bootloader. Additional stages of the bootloader are located within a filesystem (which need not be the root filesystem).

Control is passed to the kernel once it has been loaded. The kernel initialises various systems and then creates the first process (called init) which in turn, starts the system's *daemon* processes including services that allow users to login GNU/Linux systems, typically, provide a wealth of system utilities and applications which are executed as user processes.

Shared libraries are collections of object code that are linked into programs at either compile time (static libraries) or run time (dynamic libraries). There are a number of benefits to shared libraries. Library code can be modified and re-compiled without having to recompile any application software that use it. Conversely, if the application software is re-compiled, it is not necessary to re-compile any of the libraries. Furthermore, dynamic libraries reduce memory usage. Only one copy of a dynamic library need reside in memory even though many running programs may use it.

A GNU/Linux requires at least one filesystem, called the root filesystem. Much of the operating system software resides on this filesystem. The kernel and bootloader

A. Holt and C.-Y. Huang, *Embedded Operating Systems*, Undergraduate Topics in Computer Science, DOI 10.1007/978-1-4471-6603-0_2

sometimes reside on a separate (small) filesystem called the boot filesystem. One of the reason for this is, some bootloaders, such as SYSLINUX, FAT and NTFS filesystems. The bootloader and kernel will reside on a small FAT/NTFS filesystem, while the rest of the operating system resides on a native Linux filesystem (and will be the root filesystem). The GNU/Linux system resides within a filesystem.

In the chapter we present an overview of a GNU/Linux system and discuss its component parts. We also look at user processes.

2.1 The Bootloader

The bootloader is a piece of software that runs when the system boots up and is responsible for loading the kernel (amongst other things). In this section we describe the operation of the bootloader. For convenience, we assume an IBM-compatible PC system. The boot process for other architectures may vary.

The code for the bootloader (or at least the first part of the bootloader is stored in the *boot sector* of a disk (which is flagged as the boot disk). This bootsector also stores the partition table. The bootloader is either on the first block of the disk or in first block of a partition. In PC nomenclature, there are two types of boot sector:

- Master boot record (MBR): the boot sector at the start of a partitioned disk.
- Volume boot record (VBR): the boot sector at the start of a non-partitioned disk or at the start of a partition.

The MBR may contain a bootloader for booting and a kernel *or* it may contain a boot manager that *chain loads* bootstrap code in a VBR (on one of the partitions). For the purpose of this explanation, we will assume the bootloader in the MBR loads the kernel directly.

Before the bootloader can be executed, it has to be loaded from disk into memory. On IBM-compatible systems this is done by the basic input output system (BIOS). When the CPU is powered on it executes a jump instruction which passes control to the BIOS code in ROM. The BIOS initialises some of the hardware and then performs a power-on self-test (POST).

If the POST passes, the BIOS loads the content of the MBR (the bootloader code and the partition table) from the boot disk into memory at location 0x007c00. It then jumps to the first instruction of the bootloader. The bootloader loads the kernel into memory and passes control to it. As there is only 440 bytes available for bootstrap code, there is a limit to what the MBR resident bootloader can do. For this reason, the MBR bootloader loads a second stage bootloader from hard disk (which do not have the same size restrictions as the MBR bootloader). Additional bootloader stages may be loaded before the kernel itself is loaded.

There are a number of bootloaders available for booting Linux kernels. A few of the popular ones are described below:

LILO The Linux loader (LILO) was the default loader for GNU/Linux distributions in the early years. LILO has been superseded by GRUB as the default bootloader.
GNU GRUB Grand unified bootloader is a sophisticated bootloader that is used extensively with GNU/Linux systems. It comes in two flavours, GRUB 1 and GRUB 2. However, there is no longer any development for GRUB 1 and it is being phased out in favour of GRUB 2.
SYSLINUX SYSLINUX is a lightweight bootloader which is ideal for GNU/Linux embedded systems. SYSLINUX can only boot from FAT and NTFS filesystems. There are a number of SYSLINUX derivatives:

- ISOLINUX is a bootloader for ISO 9660 CD-ROM filesystems.
- PXELINUX is a bootloader for booting over the network from a remote server. This is based on the preboot execution environment (PXE) system.
- EXTLINUX is a version of SYSLINUX which can boot from ext2, ext3 and ext4 filesystems.

Das U-Boot Das U-Boot (universal bootloader) is an open source boot loader specifically for embedded devices. It is available for various architectures, for example: ARM, AVR32, MIPS, PPC, x86, 68k, Nios, and MicroBlaze.

2.2 The Kernel

The Linux kernel operates between the computer system hardware and the user application. It manages the hardware (CPU, memory and peripheral devices), and runs user programs. Computer system may be shared by multiple users so the kernel must also ensure the integrity of the system and enforce security. Linus Torvalds began development of the Linux kernel in 1991. However, since then a large community of developers have contributed to its development. Indeed, the majority of development is done by this community rather than Torvalds himself. Nevertheless Torvalds maintains an active role in the development and has the final say regarding new features. The Linux kernel is released under the GNU General Public License version 2 (GPLv2). The source code for the Linux kernel can be downloaded from [1]. The kernel source directory is divided into subdirectories, a brief summary of the structure is outlined below:

linux/kernel	The core kernel code.
linux/include	Contains header files for both kernal and user applications.
linux/arch	Architecture dependent parts of the kernel.
linux/drivers	Device driver code.
linux/fs	Code the virtual file system (VFS) and the physical filesystems.
linux/init	Architecture independent boot code and the initial entry point to kernel.

linux/ipc	Source code for interprocess communication (IPC) methods, namely, semaphores, shared memory and messages.
linux/mm	Memory management functions, namely, Paging and swapping, Allocation and deallocation of memory and Memory mapping.
linux/scripts	Supporting scripts for kernel configuration, patching and documentation.

The Linux kernel resides in the */boot* directory (typically) in either of two file formats: zImage or bzImage (big zImage). The kernel image comprises two parts, a compressed part preceded by uncompressed header. The bootloader loads the kernel image into memory and passes control to the first instruction in the uncompressed header. This part of the kernel runs in real mode (for i386 processors). After completing some initialisation tasks it switches to protected mode. The kernel relocates itself (several times) in memory before decompressing the compressed part of the image. Control is then passed to this part of the kernel. This thread of execution eventually becomes the idle process which executes when there are no user processes on the run queue.

The kernel creates the first user process and assigns it a process ID (PID) of 1. The init program code is located and executed.

The Linux kernel is highly customisable with many configuration options (such as device drivers). These options can either be built into the kernel image *or* as a loadable module. If the latter is used, an option in the kernel needs to set to support loadable modules. Modules can be added (resp. removed) to (resp. from) the kernel dynamically. The benefit of this is, they do not need to resident in memory if they are not actually required. Embedded system, however, tend to have fixed hardware configurations and, therefore, a set number of device drivers which must be loaded at all times. In which case they may as well be built into the kernel.

2.3 The Init Process

The init process is the first user space process. All processes runnning on the system are descendants of the init process (though not necessarily direct descendants). There are a number of versions of init. In this book we restrict our discussions the System V version (Sysvinit). The System V initialisation procedure adopts the concept of *runlevels*. Runlevels determine the state of the machine after boot and thus the services that are started or terminated. Examples of operating system modes defined by runlevels are:

- Single-user mode.
- Multi-user mode with no network services.
- Multi-user mode with network services.
- Shutdown.
- Reboot.

Table 2.1 Unix system V runlevels used by the Linux standard base

ID	Name	Description
0	Halt	System shutdown
1	Single-user mode	Single-user mode for administrative tasks
2	Multi-user mode	Multi-user mode, no networking
3	Multi-user mode with networking	Starts the system
4	Not used/user-definable	For special purposes
5	Multi-user mode with GUI	Same as runlevel 3 with display manager
6	Reboot	Reboot the system

Table 2.2 Unix system V runlevels used by Debian

ID	Name	Description
S		Run at boot time
0	Halt	Stop the system
1	Single-user mode	Single-user mode
2–5	Multi-user mode	Console logins and display manager
6	Reboot	Reboot the system

With GNU/Linux systems, runlevels are assigned and used differently depending upon the distribution. Table 2.1 shows how runlevels for the Linux standard base (LSB). In contrast, Table 2.2 shows how runlevels are used in Debian. We can see from the output of the command below, that the system is currently running at runlevel 2:

```
$ who -r
         run-level 2  2013-04-16 08:33
```

2.4 The Root Filesystem

The GNU/Linux file space is organised into directories. The purpose of this directory structure is to impose some logical organisation on the filesystem. The files that support the operation of the system are stored within this file space:

- Kernel and loadable modules.
- Second (and third) stage bootloaders.
- Start-up/shutdown scripts.
- Configuration files.
- System utilities.

The GNU/Linux file space may comprise several filesystems mounted at various points (directories) within the hierarchy. The root filesystem is mounted at the "/" directory (called the root directory) at the top of the hierarchy. Most of the files listed above are stored on the root filesystem. The kernel and bootloaders, however, may be stored on another filesystem. GNU/Linux supports many different filesystem type, a few are listed below:

ext2/ext3/ext4 The Extent filesystem is the default Linux filesystem. The original Extent filesystem, ext, has been deprecated and replaced with ext2, ext3 and ext4.
ReiserFS/Reiser4 A journalling filesystem. Reiser4 is the successor to ReiserFS.
SquashFS A read-only filesystem that compresses, files, directories and inodes. SquashFS is aimed at devices with limited storage, such as embedded systems.
XFS A high performance journalling filesystem. Parallel I/O can be performed on XFS filesystems.
FAT/NTFS Microsft filesystems.

The command-line below shows the directory structure for actual GNU/Linux system. For brevity we only show the hierarchy to a directory depth of two and omit any regular (or special) files:

```
# tree -dL 2 /
/
|-- bin
|-- boot
|   `-- extlinux
|-- dev
|   |-- misc
|   |-- shm
|   `-- ubd
|-- etc
|   |-- init.d
|   |-- network
|   |-- profile.d
|   |-- rc0.d
|   |-- rc1.d
|   |-- rc2.d
|   |-- rc3.d
|   |-- rc4.d
|   |-- rc5.d
|   |-- rc6.d
|   |-- rc.d
|   `-- rcS.d
|-- include
|   `-- ncurses
|-- lib
```

```
|   `-- modules
|-- proc
|-- root
|-- sbin
|-- share
|   |-- info
|   |-- locale
|   `-- man
|-- sys
|-- usr
|   |-- bin
|   |-- lib
|   |-- libexec
|   |-- sbin
|   `-- share
`-- var

39 directories
```

The GNU/Linux directory structure is defined in the filsystem hierarchy standard (FHS) and is governed by the Free Standards Group [2]. It is made up of a number of vendors, such as HP, Red Hat and IBM. The purpose of the FHS is enable software and users to predict the location of files and directories. A description of some of the main directories is given below:

/bin	Essential command binaries. These are commands that can be used by all users, that is, administrators as well as regular users. The entries in the directory can either be hard links (binaries are actually in the directory) or symbolic links (binaries in some other directory). There must not be any subdirectories under */bin*.
/boot	Bootloader files. The configuration files and further stages of the bootloader are kept under this directory, though they may be in subdirectories. The kernel file can also be in */boot* (it could be in / instead).
/dev	Device and special files.
/etc	Host-specific system configuration files.
/lib	Shared libraries and kernel modules.
/media	Removable media mount point.
/mnt	Mount point for mounting a filesystem temporarily.
/opt	Add-on application software packages.
/sbin	Essential system binaries.
/srv	Data for services provided by this system
/tmp	Temporary files.
/usr	Secondary hierarchy.
/var	Contains variable data that changes frequently during the systems operation. For example, log files and caching data.

/root Home directory for superuser.
/home User home directories.
/proc This is a virtual filesystem in which processes and kernel information is kept. In the case of the GNU/Linux operating system, the procfs filesystem is mounted here.

The filesystem will be discussed in greater details in Chap. 3.

2.5 Process Management

Processes are programs under execution. The kernel is responsible for management and scheduling of processes on the system. Processes are created using fork() system call. The fork() system call creates a replica of the process that made the call. Then one of the exec family of system calls is then used to overlay the newly created process with a new program image. A detailed description of Unix/Linux system programming is outside the scope of this book. We refer the reader to [3] for a more detailed explanation. In this section we describe processes management.

2.5.1 Signals

Signals are software interrupts that notify processes that some condition or event has occurred, for example, divide by zero, illegal instruction executed or the Interrupt key is pressed. Furthermore, signals can be sent from other processes. Processes can respond to signals in three different ways:

- Perform the default action.
- Ignore the signal.
- Catch the signal and run a function.

The default action performed by a process when it receives a signal depends on the signal type. Most (like SIGKILL) cause the process to terminate. Others like SIGQUIT cause the process to terminate and generate a core dump. A list of all the different signals can be displayed using the command:

```
$ kill -l
EXIT HUP INT QUIT ILL TRAP ABRT EMT FPE KILL BUS SEGV
SYS PIPE ALRM TERM USR1 USR2 CLD PWR WINCH URG POLL
STOP TSTP CONT TTIN TTOU VTALRM PROF XCPU XFSZ WAITING
LWP FREEZE THAW CANCEL LOST RTMIN RTMIN+1 RTMIN+2
RTMIN+3 RTMAX-3 RTMAX-2 RTMAX-1 RTMAX
```

Start a process to sleep for 300 s (five min):

```
$ ( sleep 300 )
```

If we wish to terminate the process before five min has elapsed, we can send the process a signal. Some signals can be sent using keyboard control characters. The command below shows the mapping of control characters to signals for current terminal:

```
$ stty -a
speed 38400 baud; rows 40; columns 157; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D;
eol = M-^?; eol2 = M-^?; swtch = M-^?; start = ^Q;
stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W;
lnext = ^V; flush = ^O; min = 1; time = 0;
-parenb -parodd cs8 hupcl -cstopb cread -clocal
-crtscts
-ignbrk brkint -ignpar -parmrk -inpck -istrip -inlcr
-igncr icrnl ixon -ixoff -iuclc ixany imaxbel iutf8
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel
nl0 cr0 tab0 bs0 vt0 ff0
isig icanon iexten echo echoe echok -echonl -noflsh
-xcase -tostop -echoprt echoctl echoke
```

We can see from the output above that the CONTROL-C key sequence generates an *intr* (SIGINT) signal. Press CONTROL-C to terminate the foreground process and return the shell prompt:

```
^C
$
```

If we run a process in background, we cannot send it signals using control characters from the keyboard. Instead we must use the kill command. To demonstrate, start a sleep process in background:

```
$ sleep 300 &
[1] 3603
```

The process id (PID), in this case, is 3603. We can confirm that the process is active:

```
$ ps -p 3603
  PID TTY          TIME CMD
 3603 pts/4    00:00:00 sleep
```

Send a SIGINT to the process 3603 with the kill command:

```
$ kill -SIGINT 3603
```

Now confirm process 3603 has terminated:

```
$ ps -p 3603
  PID TTY          TIME CMD
[1]+  Interrupt               ( sleep 300 )
```

Traps are used to change the default behaviour of processes in response to signals. The command-line below starts a subshell. The trap statement changes the *disposition* of SIGINT. We do not specify an action so the disposition is to ignore the signal:

```
$ ( trap '' SIGINT ; sleep 300 ) &
[1] 3941
```

Confirm process 3941 is active:

```
$ ps -p 3941
  PID TTY          TIME CMD
 3941 pts/4    00:00:00 bash
```

Send a SIGINT to the process:

```
$ kill -SIGINT 3941
```

We see that process 3941 has ignored the signal:

```
$ ps -p 3941
  PID TTY          TIME CMD
 3941 pts/4    00:00:00 bash
```

We terminate the process with SIGTERM:

```
$ kill -SIGTERM 3941
$ ps -p 3941
  PID TTY          TIME CMD
[1]+  Terminated          ( trap '' SIGINT; sleep 300 )
```

Note, if no signal is specified, the kill command sends SIGTERM (terminate signal) by default. Thus, the command-line above could have been replaced by `kill 3941`.

Some signals cannot be ignored or caught. In the example, below, we try to ignore SIGKILL:

```
$ ( trap '' SIGKILL ; sleep 300 ) &
[1] 3950
```

Regardless, the process terminates upon receiving SIGKILL:

```
$ kill -SIGKILL 3950
$ ps -p 3950
  PID TTY          TIME CMD
[1]+  Killed              ( trap '' SIGKILL; sleep 300 )
```

In this example we change the disposition to a specific action (note this time we run the subshell in foreground):

```
$ ( trap 'echo "GOTCHA!"' INT ; sleep 300 )
```

Enter the CONTROL-C key sequence (generate SIGINT) and the signal handler invokes the signals disposition action:

```
^CGOTCHA!
```

2.5.2 Job Control

If a program is run as a foreground process, it takes control of the terminal. Control is returned to the shell when the foreground process terminates. Processes can be started in background (using the & operator), in which case, the shell retains control of the terminal and the user can issue commands while the background process runs. The process can be suspended using a SIGSTOP (or SIGTSTP) signal. While this halts any further execution of the process, it does not terminate. Execution of the process can be resumed from the point it initially suspended. Start a process in foreground:

```
$ sleep 300 ; echo process ended
```

A SIGSTOP can be sent to the (foreground) process from the keyboard with a CONTROL-Z:

```
^Z
[1]+  Stopped                 sleep 300
```

The default disposition of SIGSTOP is to pause the foreground process. The command below shows the processes that are running and the states that they are in:

```
$ ps -o pid,ppid,state,wchan,comm
  PID  PPID S WCHAN  COMMAND
 4670  2155 S wait   bash
 5128  4670 T signal sleep
 5129  4670 R -      ps
```

Process 5129 is the process for the ps command we issued above. It is in a state running or runnable (R). Process 4670 is our shell process. It is in a state of interruptible sleep due to the wait system call it issued after starting the ps command. The process we are interested in is 5128, the sleep 300 command we issued earlier and suspended with SIGSTOP. We can see it is in a state T which means stopped.

2.6 Process Input/Output

When a process is created, three I/O (input/output) streams are opened, namely, standard input, standard output and standard error. By default, standard input is mapped to the keyboard while standard output and standard error are mapped to the screen. The diagram in Fig. 2.1 shows a graphical representation of a processes (initial) I/O streams.

Table 2.3 shows a summary of the I/O streams. It shows that each I/O stream is associated with a *file descriptor*. When any file object is opened by a process, a file descriptor is created which is an index to the file object. Initially, file descriptors 0,1 and 2 mapped to devices, that is, keyboard and screen. However, these file descriptors can be mapped to other file objects. These file objects can be other devices or actual

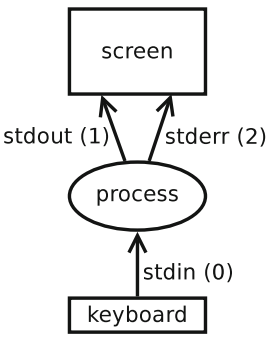


Fig. 2.1 Process input/output

Table 2.3 Summary of input/ouput streams

File descriptor	Stream name	Abbreviation	Default device
0	Standard input	stdin	Keyboard
1	Standard output	stdout	Screen
2	Standard error	stderr	Screen

regular files. This called I/O *redirection*. The shell supports redirection of the standard I/O streams. In this section we present a number of examples of I/O redirection.

The > operator is used to redirect standard output. We use the date command to demonstrate. The function of the date command is to display the current time and date on stdout. Without specifying any redirection, the time and date is displayed on the screen:

```
$ date
Mon Sep 16 15:29:15 BST 2013
```

We run the command again, however this time we redirect stdout to a file *datefile*:

```
$ date > datefile
```

We see nothing displayed on the screen. If we examine *datefile*, we see that it contains the time and date of when the date command was run:

```
$ cat datefile
Mon Sep 16 15:32:05 BST 2013
```

As *datefile* did not exist before we issued the date command, the stdout redirection operator caused the shell to create it. If *datefile* had already existed, its content would have been overwritten by the output of date. The >> operator also redirects stdout to a file, but unlike > if appends the output of a command to the current content. For example:

```
$ date >> datefile
$ cat datefile
Mon Sep 16 15:32:14 BST 2013
Mon Sep 16 15:32:34 BST 2013
```

The < operator redirects a file to stdin of a process. In the example we use the dc command which is a calculator that accepts mathematical expressions in reverse-polish notation. By default, dc reads stdin (in this case is the keyboard) for instructions. We enter the string "2 3 + p" and process outputs the answer (5):

```
$ dc
2 3 + p
5
```

To show how stdin can be redirected from a file, create a file with an expression in reverse polish notation:

```
$ echo "2 3 + p" > dcfile
```

Now run dc and redirect *dcfile* to stdin:

```
$ dc < dcfile
5
```

Standard input can also be redirected from Here documents using the << operator. A Here document is a literal input stream that appears within the source code itself. In the example below, `EOF` is used as a delimiter to indicate the beginning and end of the contents of the Here document:

```
$ dc <<-EOF
> 2 3 + p
> 10 * p
> EOF
5
50
```

Standard output and standard error are separate streams even though they both default to the screen. However, as they are separate streams, they can be redirected separately. Consider the command-line below:

```
$ time date
Tue Sep 17 11:14:35 BST 2013

real 0m0.004s
user 0m0.004s
sys  0m0.004s
```

As we know, the date command displays the current time and date. The time command displays the system resources that the date command used (in terms of real, user and system time). If we repeat the command, but this time redirect stdout, we see that the ouput of the date command has been redirected to *datefile*. The output from the time command, however, still appears on the screen. This is because the output of the time command is on stderr.

```
$ time date > datefile

real 0m0.005s
user 0m0.004s
sys  0m0.004s
$ cat datefile
Tue Sep 17 11:32:22 BST 2013
```

We can redirect stderr with the 2> operator:

```
$ (time date) 2> timefile
Tue Sep 17 11:37:32 BST 2013
$ cat timefile

real 0m0.004s
user 0m0.004s
sys  0m0.004s
```

We can see that the output of the time command (stderr) is redirected to *timefile*. The output of the date command (stdout) is unaffected and is displayed to the screen.

Both stdout and stderr can be redirected to a single file by combining both streams with the 2>&1 operator and redirecting stdout to *tdfile*:

```
$ (time date) > tdfile 2>&1$
cat tdfile
Tue Sep 17 11:40:16 BST 2013

real 0m0.004s
user 0m0.000s
sys  0m0.008s
```

It is also possible to redirect the standard output of one process to standard input of another. The | operator is called a *pipe*. Pipes are a Unix interprocess communication (IPC) mechanism [3]. The example below generates a DNS request for the NS records of the ac.uk domain. The output of the command produces a number of lines output. We filter out all but the actual NS records by piping the output of dig to the grep command:

```
$ dig ac.uk NS | grep "^ac"
ac.uk.      60177   IN      NS    ns2.ja.net.
ac.uk.      60177   IN      NS    ns0.ja.net.
ac.uk.      60177   IN      NS    ns1.surfnet.nl.
ac.uk.      6017    IN      NS    ws-fra1.win-ip.dfn.de.
ac.uk       6017    IN      NS    ns3.ja.net.
```

Table 2.4 Redirection operators

Operator	Action
<	Redirect stdin
>	Redirect stdout
>>	Append to stdout
<<	Redirect from Here document
2>	Redirect stderr
2>&1	Redirect stderr to stdout
|	Pipe stdout to stdin of another process
-2>&1|	Pipe stdout and stderr to stdin of another process

```
ac.uk       60177   IN      NS    ns4.ja.net.
ac.uk       60177   IN      NS    auth03.ns.uu.net.
```

A summary of redirection operators are shown in Table 2.4.

2.7 The Process Environment

Every process has an *environment*. The environment of a process is a NULL terminated character array list. Each element in the list is an environment variable of the form *name* = *value*. Environment variables can be created, deleted and set during the execution period of the process. This section describes the Unix process environment and the system call functions available for accessing it. The C program in Listing 2.1 declares a pointer to a character array, *env*. Each element of array is displayed to the screen.

Listing 2.1 *Contents of shenv.c*

```
#include <stdio.h>

main(int argc, char **argv, char **env)
{
    while(*env != (char *) 0)
        printf("%s\n", *env++);
}
```

Compile *shenv.c* and execute it. The program will output many lines. For brevity, we pipe it through a filter so that only the value of the SHELL variable is displayed:

```
$ gcc -o shenv shenv.c
$ ./shenv | grep SHELL
/bin/bash
```

The environment of a process can also be accessed using the global *environ* variable. The code extract below demonstrates its use:

```
char **p=environ;
while(*p != (char *) 0)
    printf("%s\n", *p++);
```

The value of individual environment variables can be accessed using the `getenv()` function. Create a file *get_env.c* with the contents shown in Listing 2.2.

Listing 2.2 *Contents of getenv.c*

```
/* get environment variables */
#include <stdlib.h>
main(int argc, char **argv, char **env)
{
    int ret;
    char *value;

    if(argc<2)
    {
         printf("usage: %s name\n", argv[0]);
         exit(-1);
    }

    if((env_val=getenv(argv[1])) < 0)
        printf("error running getenv\n");
    printf("%s\n", value);
}
```

Compile *getenv.c* then execute the resultant binary code with a variable name as an argument:

```
$ ./getenv SHELL
/bin/bash
```

Environment variables can be set with the `setenv()` function. Listing 2.3 shows the program for setting an environment variable. Compile it in the usual way.

Listing 2.3 *Contents of setenv.c*

```
/* set environment variables */
#include <stdlib.h>
main(int argc, char **argv, char **env)
{
    int ret;
    char *value;

    if(argc<3)
    {
        printf("usage: %s name value [0|1]\n", argv[0]);
        exit(-1);
    }

    value=getenv(argv[1]);
    printf("value of %s before setenv() is %s\n", argv[1], value);
    if(setenv(argv[1], argv[2], atoi(argv[3]))<0)
        printf("error with setenv function\n");

    value=getenv(argv[1]);
    printf("value of %s after setenv() is %s\n", argv[1], value);
}
```

Execute `setenv` with three arguments, that is: a variable name, the value to set the variable to and an overwrite parameter. If an existing variable is specified then it will only change to the specified value if the overwrite is 1. In the example below the variable GREET already exists. The value of GREET is unchanged after the `setenv()` function call because the overwrite parameter is 0.

```
$ ./setenv GREET hello
GREET before setenv() is (null)
GREET after setenv() is hello
```

Set the GREET variable from the command-line:

```
$ export GREET="welcome"
```

An attempt to set GREET fails because the overwrite flag is 0 (by default):

```
./setenv GREET hello
GREET before setenv() is welcome
GREET after setenv() is welcome
```

Setting the overwrite flag explicitly to a non-zero value sets the environment variable:

```
$ ./setenv GREET hello 1
GREET before setenv() is welcome
GREET after setenv() is hello
```

Environment variables can also be created and set with the function `putenv()`. The putenv() function places an entry onto the environment list in the form name = value. If the variable does not exist then it is created. If it does exist, the old entry is replaced. Create a file *putenv.c* with the contents shown in Listing 2.4.

Listing 2.4 *Contents of putenv.c*

```
/* put environment variables */
#include <stdlib.h>

main(int argc, char **argv, char **env)
{
    int ret, i=0;
    char *value, name[32], *p;
    if(argc<2)
    {
         perror("usage: %s name value [0|1]\n", argv[0]);
         exit(1);
    }

    for(p=argv[1]; *p != '='; p++, i++)
        name[i]=*p;
    name[i]=(char) 0;

    env_val=getenv(env_name);
    printf("value of %s before putenv() is %s\n", name, value);

    if(putenv(argv[1]) != 0)
        perror("error with putenv function\n");

    value=getenv(name);
    printf("value of %s after putenv() is %s\n", name, value);

}
```

Compile *putenv.c* then execute the binary code:

```
$ putenv GREET=welcome
value of GREET before putenv() is hello world
value of GREET after putenv() is welcome
```

Child processes do not inherit environment variables unless they are *exported*. Run shenv and piped through a filter.

```
$ ./shenv | grep GREET
```

The execution of the command-line above results in no output. This is because the variable GREET has not been exported in the shell and is not inherited by the child process (in this case shenv). Export the variable GREET:

```
$ export greeting
$ ./shenv |  grep GREET
GREET=hello
```

2.7.1 Shell Parameters

The shell, like any other Unix process has an environment. The shell's environment can be accessed through its parameters from the command-line. There are three types of parameter:

- named parameters
- positional parameters
- special parameters

Named parameters are known as environment variables. Some of these variables are initialised when the user logs in. The shell uses these variables to determines its behaviour. The shell also maintains certain variables throughout the login session, some of which are defined by the shell and are given special meaning, while others can be user defined.

Positional parameters enable reference to arguments supplied to shell scripts. The shell shares process information through special parameters. Special parameters can be accessed by the user, but unlike named and positional parameters, they can only be set by the shell itself.

Named parameters are assigned using the command line statement in the form name = value. Variable names are alphanumeric strings, they can contain numeric characters or underscores but must begin with an alphabetic character. Four examples are presented below:

```
$ n=2
$ n2=4
$ N=3
$ MYNAME=alan
$ greeting="hello world"
```

Note that Unix is case sensitive, so the variable "name" is not the same as "Name". A variable can be expanded (evaluated) by preceding the variable name with a `$`, for example:

```
$ echo $greeting
hello world
```

Variables can be concatenated:

```
$ day=21 month=01 year=2006
$ echo $day$month$year
21012006
```

However, there can be a problem with concatenating literals with variables. The example below uses variables for the day and month but hard-codes the year:

```
$ echo $day$month2006
05
```

The problem is that the shell cannot distinguish between the variable name month and the literal value 2006. It therefore tries to expand a variable with the variable name `month2006`, which is not set. This can be rectified by surrounding the variables names with braces `{}`, for example:

```
$ echo ${day}${month}2006
05112006
```

Although the braces are not always necessary, it is good practice to use them when evaluating variables.

2.7.2 Quoting

Consider the following example, where the variable expansion phrase is enclosed in double quotes:

```
$ echo "echo $greeting", displays $greeting
echo hello world, displays hello world
```

Variables enclosed in double quotes are still expanded by the shell. To prevent expansion, variables should be enclosed in single quotes. To display the string "echo $greeting" verbatim, use the statement:

```
$ echo 'echo $greeting', displays $greeting
echo $greeting, displays hello world
```

2.7.3 Positional Parameters

Positional parameters provide a means of accessing arguments to shell scripts. They are referenced by the numbers 1,2,3,…etc, where the numbers reflect the order in which the arguments appeared on the command line (from left to right). Positional parameters cannot be assigned in the same way as named parameters. For example you cannot do this:

```
$ 1=hello
1=hello: command not found
```

Positional parameters are assigned when a shell script is invoked with command line arguments. They can, however, be assigned with the set command:

```
$ set `uptime`
```

This causes the space separated strings from the output of uptime to be assigned to the positional parameters. The statement below shows the values of all positional parameters that were assigned:

```
$ echo $*
Linux underworld 2.6.32-26-generic-pae #48-Ubuntu SMP
Wed Nov 24 10:31:20 UTC 2010 i686 GNU/Linux
```

The command-line below gives the number of assigned positional parameters (equivalent to argc in C):

```
$ echo $#
13
```

If we specifically want the kernel version, then we evaluate the third positional parameter:

```
$ echo $3
2.6.32-26-generic-pae
```

Double digit positional parameters need to be enclosed in braces:

```
$ echo ${12}
i686
```

Finally, positional parameter 0 is set to the name of the current process (equivalent to `argv[0] in C`), which in this case should be the name of the shell command itself:

```
$ echo $0
bash
```

2.7.4 Special Parameters

Special parameters can be accessed by the user but can only be assigned (directly) by the shell, typically in response to some event. We have already encountered the `*` and `#` parameters which are set when a command is invoked with arguments (or with the set command).

The `?` parameter expands to the return code of the process that last exited. If we enclose a command line statement in brackets it is executed in a subshell. The command line below starts a new shell and immediately issues exit. The argument following exit is the value of the subshells return code (255 in this case):

```
$ (exit 255)
```

The return code of the child process can be accessed by the parent through the `?` parameter. If a process terminated normally then the return code is likely to be zero. If however, the process terminated due to some error condition, it is useful if process sets a non-zero return code so that the parent can determine the reason for the termination:

```
$ echo $?
255
```

The process ID of the shell is stored in the `$` parameter. The ps command shows that the process ID of the foreground shell is 10005, which is what is given by the expansion of `$`:

```
$ ps | grep bash
10005 pts/77            00:00:00 bash
$ echo $$
10005
```

The ! parameter evaluates to the process ID of the last background process. To demonstrate, the statement below invokes the sleep command (which sleeps for 60 s) in the background:

```
$ sleep 60 &
[1] 24835
```

The process ID of the background process is displayed on the screen (in this case 24835). Expansion of the ! parameter confirms this:

```
$ echo $!
24835
```

As we saw in the section above. processes do not inherit environment variables from their parent unless the variables are exported. Here, we explore shell variable exportation further. The command line below displays the process ID of the current process followed by the value of greeting (which was assigned earlier):

```
$ echo ${greeting?parameter not set}
hello world
```

If we start a new shell and issue the command again we see that greeting is not set in the child process:

```
$ bash
$ echo ${greeting?parameter not set}
bash: greeting: parameter not set
```

Terminate the current shell process and return to the parent, then export greeting:

```
$ exit
$ export greeting
```

You can confirm that a variable has the export attribute set by typing:

```
$ export | grep greeting
declare -x greeting="hello world"
```

Now start a new shell again and evaluate greeting:

```
$ bash
$ echo ${greeting?parameter not set}
hello world
```

The new shell has inherited the variable greeting. A variable retains its export attribute (until the shell is terminated) even if it is reassigned, but it can be removed using the typeset command:

```
$ typeset +x greeting
$ export | grep greeting     # yields no result
```

In Bash (but not the original Bourne shell), a variable can be assigned and exported in one command line statement:

```
$ export greeting="hi there"
$ export | grep greeting
declare -x greeting="hi there"
```

2.8 Summary

In this chapter we discussed the component parts a GNU/Linux system, namely the bootloader, kernel and filesystem. In later chapters we will describe in details to how build these components.

Once the system has booted (init process has been initialised), system daemons and user applications are run as processes, where processes are programs under execution. Processes are managed by the kernel. We covered process management (signals and job control), input/output and the process environment.

References

1. Linux Kernel Organization, Inc. (2013) The linux kernel archives. https://www.kernel.org/. Accessed 22 July 2013
2. Filesystem Hierarchy Standard Group (2013) Filesystem hierarchy standard. http://refspecs.linuxfoundation.org/FHS_2.3/fhs-2.3.pdf. Accessed 24 July 2013
3. Haviland K, Salama B (1987) Unix system programming. Addison-Weseley, Boston

The Filesystem in Detail

3

Persistent data is stored in files. Files may contain text, database records, source code or executable instructions but as far as the kernel is concerned, the content of a file is merely an unstructured byte stream. Files are organised in *directories*. Directories themselves are files, however, unlike "regular" files the kernel imposes a structure to their content. Directories contain a list of *links* to files (and other directories) in the filesystem. A collection of files and directories, along with associated *meta data* form a *filesystem*.

Files that store user data are sometimes called *regular files* in order to distinguish them from other file objects in the filesystem. In addition to regular files and directories, other file objects can be accessed through the filesystem:

Device files A device file or *special* file is a device driver interface. Devices files can be accessed through standard I/O calls in the same way as a regular file even though hardware the driver controls, may not be a storage device.
FIFO FIFOs (first in first out) or named pipes are file objects used for interprocess communication (IPC).
Unix domain sockets A Unix domain socket is an IPC mechanism similar to a FIFO. Sockets are created and managed with same systems calls used for network sockets.
Symbolic links Symbolic links or soft links overcome the limitation of hard links which cannot link to files across filesystem boundaries.

A filesystem is divided into blocks (typically between 512 and 4,096 bytes in size). Blocks are allocated for storing the content of files and directories. Some blocks are reserved for meta data and information about the filesystem itself.

Physical disk drives can be divided into (one or more) *partitions* and filesystems are allocated to partitions.

In this chapter we will examine the filesystems in detail. We discuss the organisation of physical disks and how filesystems are mapped to disk partitions.

A. Holt and C.-Y. Huang, *Embedded Operating Systems*, Undergraduate Topics in Computer Science, DOI 10.1007/978-1-4471-6603-0_3

3.1 GNU/Linux File Space

While primary storage (main memory) is accessible directly from a program, secondary storage is only accessible through kernel services. The part of the kernel that provides these services is called the filesystem. The purpose of the kernel filesystem services is to present a common interface to the file space such that the complexities of the hardware are hidden from the programmer/user:

- Unix operating systems can implement the file space over a number of separate filesystems.
- Unix filesystems are sometimes referred to as *demountable volumes* [1] as they can be dynamically added and removed from the file space.
- GNU/Linux supports many different filesystem types using a virtual filesystem (VFS).
- GNU/Linux supports a variety of physical storage devices such as hard disks and solid state devices.
- Physical storage devices can be partitioned to support multiple filesystems.

We will discuss filesystems, VFS and partitions in more detail later in the chapter. In this section we will focus on the file space as viewed by user.

From the users' perspective, the structure of the filesystem and organisation of disk partitions is transparent. Data is stored in files. We sometimes refer to files that store data as *regular* files because the GNU/Linux supports other types of file object.

Even a small embedded GNU/Linux system will have a large number of files. It is important then, that the filesystem provides some method of organising files. Unix uses the concept of a *directory* to group files. Not only can regular files be grouped into directories but also other directories (subdirectories). The Unix file space, therefore, resembles an inverted hierarchical tree. Below, we introduce some Unix terms regarding directories:

Root directory The directory at the top of the file space hierarchy is called the root directory which is referenced by a forward slash "/".
Current/working directory User's work within a directory. This is called the *current* or *working* directory. The current directory can be changed at any time during the user's session (typically with the cd command). Every directory has a directory entry which is a link to itself called "." which is a way of referring to the current directory. It is useful when using commands like cp (copy) which requires both a source file and a destination file/directory as parameters. For example, if we need to copy a file into the current directory, we use the command-line:

```
$ cp /tmp/tmpfile.txt.
```

Parent directory Every directory has a directory entry called ".." which is (with the exception of root) a link to the directory above. Root does have ".." directory entry but it is merely a link to itself (just like ".").

Home directory Each user is assigned a home directory. When a user first logs in the current directory is set to the home directory.

As we have seen, file objects are given names. A file object must be unique within its own directory, but need not be unique throughout the entire file space. A file, therefore, is referenced by both its name and its location within the file space. Files are located using a list of directory names which describes a "path" through the directory hierarchy to the file. This list of directories followed by the file name itself is called the *pathname* of the file. Pathnames can either be absolute or relative. Absolute pathnames begin with a "/" and describe the path to the file from the root directory. Relative pathnames, on the other hand, do not start with a "/" and describe a path to a file from the user's current directory. For example, the kernel header file *ipv6.h* is located in the directory */usr/include/linux*. Note that the "/" is also used as a delimiter between directories. We can reference the file *ipv6.h* by its absolute pathname:

```
$ ls /usr/include/linux/ipv6.h
/usr/include/linux/ipv6.h
```

The absolute pathname is the same regardless of the user's current directory. Relative directory references, however, depend upon the user's current directory. For example, make */usr* the current directory:

```
$ cd /usr
```

Reference *ipv6.h* using a relative pathname:

```
$ ls include/linux/ipv6.h
include/linux/ipv6.h
```

If we were in our home directory (which is, say */home/aholt*), the relative pathname to the *ipv6.h* is different:

```
$ cd      # change to home directory
$ ls ../../usr/include/linux//ipv6.h
../../usr/include/linux//ipv6.h
```

Notice how we use the ".." notation to reference the parent directory. The choice of absolute or relative pathnames depends which is most convenient to use. Clearly, in the example above, an absolute pathname is fewer keystrokes than its relative pathname.

3.1.1 Permissions

Permissions determine how users can access files. A user may be entitled to read, write or execute a file, depending upon the type of user they are and the permissions that apply to them. Unix defines three types of user with respect to file access:

Owner Each file has one and only one owner. Typically, this is the user that created the file, however, it is possible to change the file's ownership.
Group User can belong to various groups which are defined in the file */etc/group*.
Other Users that are neither the owner of the file nor a member of its group can access the file according to the "other" permissions.

Permissions set for the owner override those for the group and group permission override those for other. Owners and groups have a designated user (owner) and group ID which are called the uid and gid respectively. The command-line below shows the user name and uid along with the group name and gid:

```
stat -c'%N user: %U/%u group: %G/%g' ~/.profile
'/home/aholt/.profile' user aholt/1000 group aholt/1000
```

Each user type (owner, group or other) are assigned permissions, namely, read, write and execute. A permission is either enabled if the corresponding permission bit it set to 1. Similarly it is disabled if the permission bit is set to 0. File permissions are represented as a 9-bit pattern of an integer field (in the file's inode). The least significant three bits are the other users permission (read, write and execute). The three most significant bits are the owners permissions and the middle three bits are for the group. Each three bit grouping yields an octal value which describes the files permissions for the respective user type. Table 3.1 shows the octal values of each permission.

Table 3.1 File permissions

Octal value	Comment
0001	Other can execute
0002	Other can write
0004	Other can read
0010	Group can execute
0020	Group can write
0040	Group can read
0100	Owner can execute
0200	Owner can write
0400	Owner can read

We demonstrate file permissions with an example. Create a temporary file:

```
$ touch /tmp/tmpfile
```

Examine the permissions of the file:

```
$ stat -c'%a %A' /tmp/tmpfile
644 -rw-r--r--
```

We can see that read and write permissions are set for the owner ($0400 + 0200 = 0600$) and only read permissions are granted for the group and other (0400).

In this example we add execute permission to the owner (even though it contains no executable content), write permission for the group and remove all permissions for other. Setting all the permission bits on for the owner we calculate the octal value: $0400 + 0200 + 0100 = 0700$. Setting read and execute permission bits on for the group octal value is: $0400 + 0100 = 0500$. For other, all the permission bits are cleared, therefore, the value is change the permissions:

```
$ chmod 750 /tmp/tmpfile
```

We verify the permissions with the stat command:

```
$ stat -c'%a %A' /tmp/tmpfile
750 -rwxr-x---
```

3.1.1.1 Other File Objects

Unix operating systems adopt a philosophy "everything is a file". As mentioned earlier, file objects are not just confined to regular files and directories. In this subsection we will discuss, with examples, the other file objects found within the filesystem. The list below serves as a reminder of these file objects:

- Device files (special files)
- FIFOs (named pipes)
- Unix domain sockets
- Symbolic links.

3.1.1.2 Device Files

Device files are input/output interfaces to device drivers. Just like regular files, device files have directory entries and inodes but unlike regular files, devices are not allocated blocks on any disk (other than for their inode).

There are two type of device files, namely, character and block. Character devices are devices which send and receive raw streams of data. Serial lines and printers are examples of character devices. Whereas a character devices can transfer data one character at time, block devices transfer data in blocks. Block devices are typically file storage devices, for example hard disks or solid state devices. The command-line below shows examples of both types:

```
$ stat -c'%N %F' /dev/ttyS0 /dev/sda1
'/dev/ttyS0' character special file
'/dev/sda1' block special file
```

The */dev/ttyS0* device is the host serial port and is, therefore, a character device. The */dev/sda1* is a block device because it is a filesystem partition. Devices files also have *major* and *minor* numbers. The major number identifies the device driver type and the minor number specifies the instance of the device. For example, */dev/ttyS0* and */dev/ttyS1* are both serial devices and have the same major number, which is 4 (because the same driver code is used to control both devices). They have minor number of 40 and 41 respectively in order to distinguish between the two interfaces:

```
$ stat -c'%N major: %t, minor: %T' /dev/ttyS[01]
`/dev/ttyS0' major: 4, minor: 40
`/dev/ttyS1' major: 4, minor: 41
```

In this subsection we will demonstrate the concept of device file by writing a rudimentary device driver. We will write the driver as a loadable module. Listing 3.1 shows the source code for the dummy device driver *ddev.c*. It comprises two functions: dd_init() and dd_cleanup() which are run when we load the module and unload the module respectively.

Listing 3.2 shows the source code for the include file *dd_fops.h*. This defines the functions that enable the programmer to interact with the driver, albeit that interaction is limited as only open and close operations are defined. Nevertheless, this is sufficient to demonstrate how we can access a device file just as we would with any regular file.

Listing 3.1 *ddev.c*

```
/* ddev.c */

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

#include"dd_fops.h"
```

```
/* dummy device initialisation function */
int dd_init(void)
{
    int ret;


    ret = register_chrdev(dd_major, DEV, &dd_fops);
    if (ret < 0) {
        printk(KERN_WARNING \
                    "ddev: unable to assign major\n");
        return dd_major;
    }

    if (dd_major==0) dd_major = ret;

    printk(KERN_DEBUG \
               "assigned major number %d\n", dd_major);
    printk(KERN_DEBUG \
                 "initialised dummy device driver\n");

    return 0;
}

/* dummy device cleanup function */
void dd_cleanup(void)
{
    unregister_chrdev(dd_major, DEV);
    printk(KERN_DEBUG \
                    "removing dummy device driver\n");
}


module_init(dd_init);
module_exit(dd_cleanup);

MODULE_LICENSE("GPL");
```

Listing 3.2 *dd_fops.h*

```
/* dd_fops.h */

#define DEV"ddev"

static int dd_major = 256;
```

```
static int dd_minor;

/* dummy device open function */
static int dd_open(struct inode *in, struct file *fp)
{
    dd_minor = (MINOR(in->i_rdev)&0x0f);
    printk(KERN_INFO"ddev: open %d\n", dd_minor);
    try_module_get(THIS_MODULE);
    return 0;
}

/* dummy device close function */
static int dd_close(struct inode *in, struct file *fp)
{
    module_put(THIS_MODULE);
    dd_minor = (MINOR(in->i_rdev)&0x0f);
    printk(KERN_INFO "ddev: closed %d\n", dd_minor);
    return 0;
}

struct file_operations dd_fops = {
    .open =  dd_open,
    .release = dd_close
};
```

We compile the ddev module using the make utility. Listing 3.3 shows the content of *Makefile*.

Listing 3.3 *Makefile*

```
obj-m   := ddev.o
KDIR    := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

all:
$(MAKE) -C $(KDIR) M=$(PWD) modules

clean:
$(MAKE) -C $(KDIR) M=$(PWD) clean
```

Compile the module by running make:

```
$ make
make -C /lib/modules/2.6.32-26-generic-pae/build
M=/home/aholt/development/lbook/fs modules
make[1]: Entering directory
'/usr/src/linux-headers-2.6.32-26-generic-pae'
  CC [M]  /home/aholt/development/lbook/fs/ddev.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/aholt/development/lbook/fs/ddev.mod.o
  LD [M]  /home/aholt/development/lbook/fs/ddev.ko
make[1]: Leaving directory
'/usr/src/linux-headers-2.6.32-26-generic-pae'
```

A successful compilation produces the module file *ddev.ko* which can be loaded into the kernel:

```
$ sudo insmod ddev.ko
```

Confirm the module has been loaded:

```
$ lsmod | grep ddev
ddev                    1412  0
```

Check the end of the kernel buffering to see the messages output by the ddev module:

```
$ dmesg | tail -n2
[20268.989854] assigned major number 256
[20268.989856] initialised dummy device driver
```

Find the assigned major number:

```
$ grep ddev /proc/devices
256 ddev
```

We create two instances of ddev in the */dev* directory. While they have the same major number (same driver code) they have different minor numbers:

```
$ sudo mknod /dev/ddev0 c 256 0
$ sudo mknod /dev/ddev1 c 256 1
```

Next we write a utility, ddio, to run in user space to access the driver. The only system calls the ddio command make are to open and close because these are the only operations ddev supports. The source code for *ddio.c* shown in Listing 3.4.

Listing 3.4 *Source code for ddio.c*

```
#include <fcntl.h>
#include <stdio.h>

main(int argc, char *argv[])
{

    int fd;

    if((fd = open(argv[1], O_RDONLY)) < 0) {
        perror("error opening file");
        return -1;
    }

    printf("hit any key to close %s\n", argv[1]);
    getchar();

    close(fd);
}
```

Compile *ddio.c*:

```
$ gcc -o ddio ddio.c
```

Use ddio to open the device driver

```
$ ./ddio /dev/ddev0
hit any key to close /dev/ddev0
```

The ddio command opens *dev/ddev0* and then waits for a response from the user. Meanwhile the *dev/ddev0* remains open. Check dmesg (in another terminal window) to see he kernel output:

```
$ dmesg | tail -n1
[20333.467439] ddev: open 0
```

In the window running ddio, hit any key to close */dev/ddev0*, then check dmesg:

```
$ dmesg | tail -n1
[20374.512204] ddev: closed 0
```

Finally unload the module:

```
$ sudo rmmod pport
```

The rmmod command will cause the cleanup function to run. We will leave it to the reader to check dmesg to verify that the appropriate messages have been written to the kernel console.

This exercise demonstrates that even though device files are different to regular files they can still be accessed using common system calls. The `open()` and `close()` systems call in *ddio.h* are called in the same way irrespective of whether the file object is a device or regular file. While ddev only supports open and close operations it could be expanded to support the other file system calls.

3.1.1.3 FIFOs and Unix Domain Sockets

FIFOs and Unix domain sockets are similar concepts so we will describe them together in this subsection. FIFOs are sometimes called named pipes because they appear as a file object within the filesystem, and so, must have a name. Unlike "unnamed" pipes, FIFOs exist after the process that created them has been terminated. For example, create a FIFO *mypipe* with the command-line below:

```
$ mkfifo mypipe
```

Confirm the file type with the stat command:

```
$ stat -c'%F' mypipe
fifo
```

Write some data to the pipe with:

```
$ echo"hello" > mypipe &
[1] 3835
```

The write process above is run in background and blocks because there is no read process (yet). In order to demonstrate the first-in-first-out nature of the FIFO we execute another process (in background) to write to the pipe:

```
$ echo "world" > mypipe &
[2] 3837
```

Now we have two process blocked in I/O. We execute a process to read from the FIFO:

```
$ cat < mypipe
hello
world
[1]-  Done                    echo "hello" > mypipe
[2]+  Done                    echo "world" > mypipe
```

Note that the data is read in the order it was written and that the write processes that were previously blocked complete their I/O calls (and eventually terminate).

Sockets are an API primarily designed for network applications where a client on one machine can communicate with a server on another. However, sockets also supports "local" sockets which are known as Unix domain sockets. We will demonstrate Unix domain sockets by writing a simple echo server (and client) in Python. The server program is shown in Listing 3.5. It waits for a client to initiate a connection over a Unix domain socket, which in this case, has the pathname */tmp/usocket*. Once a session has been established the server listens for any data on the socket. Any data it receives it "echoes" back to the client.

Listing 3.6 shows the Python client script. The client initiates a session to the server over (*/tmp/usocket*) and sends some data. It then waits for it to be sent back by the server.

Listing 3.5 *Echo server: Source code for echod.py*

```
#!/usr/bin/env python

import sys,os
from socket import *

usage ="usage: %s <socketename>" % (sys.argv[0])

s ="/tmp/usocket"

if __name__=='__main__':

    u = socket(AF_UNIX, SOCK_STREAM)
    try:
        os.unlink(s)
    except OSError:
        pass

    u.bind(s)
    u.listen(1)

    c,a = u.accept()

    while 1:
```

```
            m = c.recv(1024)
            if not m: break
            c.send(m)

        c.close()
```

Listing 3.6 *Echo client: source code for echo.py*

```
#!/usr/bin/env python

import sys
from socket import *

usage ="usage: %s <string>" % (sys.argv[0])

s ="/tmp/usocket"

if __name__=='__main__':

    if sys.argv[1:] == []:
        print usage
        exit(-1)

    u = socket(AF_UNIX, SOCK_STREAM)
    u.connect(s)
    u.send(" ".join(sys.argv[1:]))
    r = u.recv(1024)
    u.close()
    print "Received: %s" % (repr(r))
```

Run the echo server as a background process:

```
$ ./echod.py &
[1] 4348
```

This will create a socket file object, which we can see with the stat command:

```
$ stat -c'%F' /tmp/usocket
socket
```

Run the echo client supplying a message we wish to send (and receive back) from the server:

```
$ ./echo.py hello world
Received: 'hello world'
[1]+  Done                    ./echod.py /tmp/usocket
```

3.1.1.4 Symbolic Links

A directory entry constitutes a link to a file. Furthermore, it is perfectly valid for a file to have multiple links. While links pointing to the same file can be in different directories, they must be on the same filesystem. It is not possible to create a *hard* link on one filesystem to a file that resides on some other filesystem. Using the */proc/uptime* file as an example, try to create a link to it in the current directory:

```
$ ln /proc/uptime uptime
ln: creating hard link 'uptime' => '/proc/uptime':
Invalid cross-device link
```

This generates an error message indicating the command failed. This is because the current directory and */proc/uptime* exist on different filesystems. A symbolic link, on the other hand, *can* be created across filesystems:

```
$ ln -s /proc/uptime uptime
```

If we compare the inode numbers of the symbolic link *./uptime* and the file /emph/proc/uptime we see the values are different:

```
$ ls -li uptime /proc/uptime
4026531982 -r--r--r-- 1 root  root   0 2013-05-01 14:10
/proc/uptime
   3944833 lrwxrwxrwx 1 aholt aholt 12 2013-05-01 14:09
uptime -> /proc/uptime
```

This is because *./uptime* is a separate file object to /emph/proc/uptime. If *./uptime* was a hard link (assuming it could be created) then it would have the same inode number as /emph/proc/uptime.

The file types reported by the stat command, are:

```
$ stat -c'%F' uptime /proc/uptime
symbolic link
regular empty file
```

We can now access *proc/uptime* using the link *.uptime* in our current directory:

```
$ cat uptime
9928.96 17606.88
```

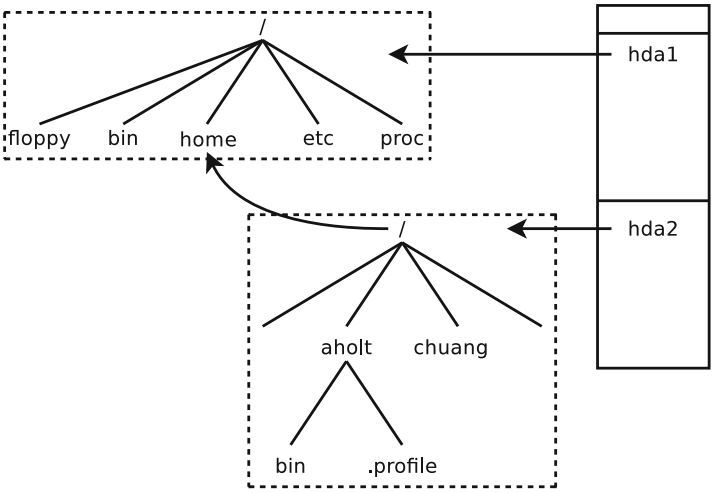


Fig. 3.1 Mounting filesystem

3.2 The Filesystem

The GNU/Linux file space comprises one or more filesystems. Unlike some operating systems, where separate filesystems are accessed by unique device identifiers, GNU/Linux filesystems are combined into a single hierarchy. GNU/Linux needs at least one filesystem which is mounted on the root directory and is referred to as the root filesystem. Subsequent filesystems can be mounted on a subdirectory. The content of any subdirectory that is used as a filesystem *mount point* is overlayed with the content of the directory at the "root" the filesystem.

The diagram in Fig. 3.1 shows partition */dev/hda1* mounted on */*. The mount operation makes the root directory of */dev/hda1*, the root directory of the file hierarchy. Figure 3.1 also shows a second partition, */dev/hda2*, mounted on */home*. The root directory of the filesystem on the */dev/hda2* device becomes the */home* directory in the file hierarchy.

GNU/Linux supports over 50 types of filesystem. In order to support multiple filesystem, Linux implements a virtual filesystem (VFS). The VFS is an abstraction layer above the actual filesystem (see Fig. 3.2). Users can then access files using a common interface regardless of structure of the underlying filesystem.

The default filesystem for GNU/Linux is the Extended filesystem. The original extended filesystem (ext) has been deprecated and ext2, ext3 and ext4 are used instead. From now on, when we use the generic term "ext" to mean either ext2, ext3 or ext4. The original Unix filesystem suffered scalability issues as the size of hard disks increased. High latencies resulted from disk heads moving between inodes at the start of filesystem and data blocks located at the end. The Berkeley fast filesystem (FFS) divides the filesystem into chunks, so that inodes and associated data blocks are located in the same general proximity on the disk. This helps to minimise head

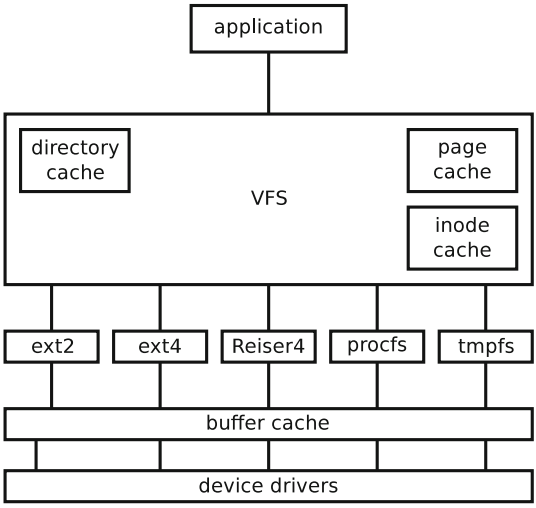


Fig. 3.2 The virtual filesystem

movement and access latency. In fact the Berkeley FFS is now considered to be the Unix filesystem (UFS) rather than the original Unix filesystem. While the original Unix filesystem is deprecated, it is worth a short review as a number of the concepts are still applicable to the ext filesystem.

The Unix filesystem is divided into equal size blocks (between 512 K and 4 M). These blocks are used to store information about the filesystems itself as well as user data. The diagram in Fig. 3.3 shows the blocks of the filesystem grouped into four areas. These areas are described below:

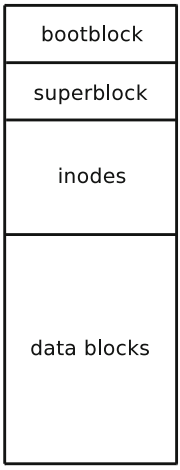


Fig. 3.3 Organisation of the Unix filesystem

Boot block The bootblock stores a small program called a *bootloader*. This program is used to load the kernel. It is loaded into memory and execution is passed to it when the system is booted.
Superblock Three types of information are stored in the superblock:

- Parameters which are fixed at the time of the filesystem's creation.
- Tunable parameters.
- Current state information.

Inodes Index nodes (inodes) are data structures which that contain meta data about file objects including pointers to the data blocks allocated to it.
Data blocks The remaining blocks store are allocated to store file content (directories as well as regular files).

The inode is the link between a file's name and the blocks on the disk allocated to it. The inode contains the file object's meta data and pointers to the data blocks on the disk where the content is stored. In order to reference a file object, its directory entry must be fetched from the directory in which it resides. The diagram in Fig. 3.4 shows a directory entry for the file *largefile.dat*. The inode number identifies the relevant inode on the disk. When the file is opened the inode data structure in the disk is loaded into memory. Figure 3.4 also shows the structure of the inode (albeit very much simplified). Some of the inode fields point directly to the file data blocks. For very large files, however, there are insufficient direct pointer fields in the inode. For this reason, some inode fields are used as indirect pointers. An indirect pointer field points to a block of data which contains pointers to either data blocks. These are called single indirect pointers, however, it is also possible to have double or even triple indirect pointers, see Fig. 3.4.

The ext filesystem is organized into block groups, analogous to cylinder groups in UFS. This is done to reduce external fragmentation and minimize the number of disk seeks when reading a large amount of consecutive data. Each block group contains a superblock, a group descriptor, a block group bitmap, an inode bitmap, followed by the actual data blocks. Backup copies of the superblock are made in every block group. However, only the first copy of it, which is found at the first block of the file system, is used. Figure 3.5 shows the organisation of an ext2 filesystem.

We can use a regular file to emulate a filesystem. Create a 64 Mbyte file with the dd command:

```
$ cd /tmp
$ dd if=/dev/zero of=testfs bs=1M count=64
```

Format the file as an ext2 filesystem:

```
$ mke2fs -F testfs
mke2fs 1.41.11 (14-Mar-2010)
```

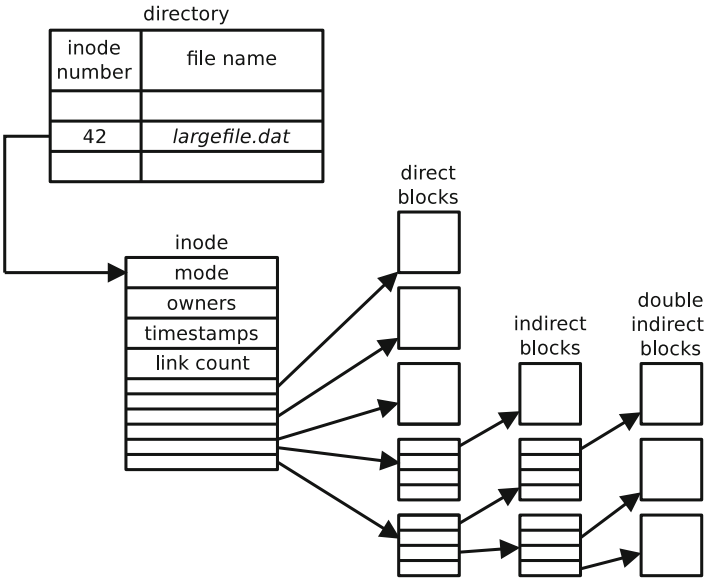


Fig. 3.4 Unix inodes and disk blocks

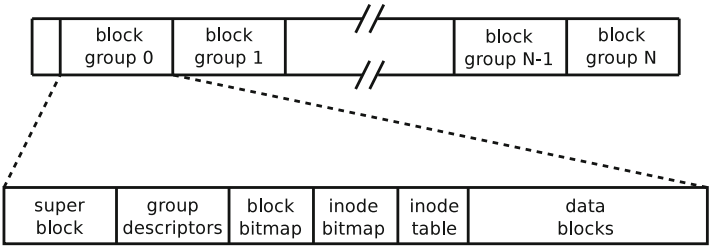


Fig. 3.5 The ext2 filesystem

```
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
Stride=0 blocks, Stripe width=0 blocks
16384 inodes, 65536 blocks
3276 blocks (5.00%) reserved for the super user
First data block=1
Maximum filesystem blocks=67108864
8 block groups
8192 blocks per group, 8192 fragments per group
2048 inodes per group
Superblock backups stored on blocks:
```

```
8193, 24577, 40961, 57345

Writing inode tables: done
Writing superblocks and filesystem accounting
information: done

This filesystem will be automatically checked every
32 mounts or
180 days, whichever comes first.  Use tune2fs -c or
-i to override.
```

Create a mount point and mount *testfs* as a loop device:

```
$ mkdir mnt
$ sudo mount -o loop testfs mnt
```

If we look at the top level directory of the filesystem mounted on *mnt*, we see just one directory entry for *lost+found* (which is itself a directory):

```
$ ls mnt
lost+found
```

Details of the filesystem can be viewed with the following command-line:

```
$ stat -f mnt
  File:"mnt"
    ID: 10300c472f0b2cc8 Namelen: 255     Type: ext2/ext3
Block size: 1024       Fundamental block size: 1024
Blocks: Total: 63461      Free: 62172      Available: 58896
Inodes: Total: 16384      Free: 16373
```

3.2.1 Pseudo Filesystems

GNU/Linux support a number of pseudo filesystems. Pseudo filesystems resemble regular filesystems in that they are demountable devices and have a directory based hierarchical structure. However, they do not map to file storage in the traditional sense. The types of pseudo filesystem are listed below:

- Tmpfs
- Procfs
- Sysfs

Tmpfs is a temporary file storage that uses main memory rather than an external storage device. Files created on a tmpfs are, therefore, non-persistent across system reboots. Examples of where tmpfs is used are:

- /tmp The */tmp* holds temporary files. It is good practice to use a separate filesystem rather than root itself. While persistent storage could be used for */tmp*, it is common for GNU/Linux systems (embedded of otherwise) to use tmpfs.
- /dev/shm Tmpfs is mounted on the */dev/shm* directory which is used for shared memory. The standard C library, Glibc, expects tmpfs to be mounted on */dev/shm*.
- /var The endurance of solid state devices is not as high as conventional disk drives. It is important, therefore, to keep write operations to the device to a minimum (some embedded systems even mount the root filesystem read-only). The */var* directory, however, *must* be read-write, so for embedded systems that use solid state devices, tmpfs can be used for the */var* directory.
- udev Traditionally, devices files in */dev* were static. However, it is common for modern GNU/Linux systems to use udev to manage the device files in */dev*. Device files are created dynamically according to udev rules. When udev is used, a tmpfs is mounted on */dev*.

The Proc filesystem (procfs) is an interface to the data structures in kernel. It exposes details about the kernel and running processes to userspace. Typically, procfs is mounted on the */proc* directory and the data structures are grouped into directories similar to the hierarchical structure of a regular filesystem. Procfs provides information about processes and other system information. The concept of sysfs is similar to procfs except that it exposes details of devices to userspace. Sysfs is mounted on */sys*.

A detailed description of procfs and sysfs is beyond the scope of this book. We, therefore, present a few examples of their use. If we want to know the TCP congestion control algorithm used by the system, we can view the file below:

```
$ cat /proc/sys/net/ipv4/tcp_congestion_control
cubic
```

View the MAC address of Ethernet interface eth0:

```
$ cat /sys/class/net/eth0/address
00:1c:23:0c:c7:53
```

Most of the files in procfs are read only. Some, however, are writeable. For example, if you need to enable IP packet forwarding, write the value 1 to the file below:

```
$ sudo echo 1 > /proc/sys/net/ipv4/ip_forward
```

We will make use of this feature in the chapters below when we configure networking.

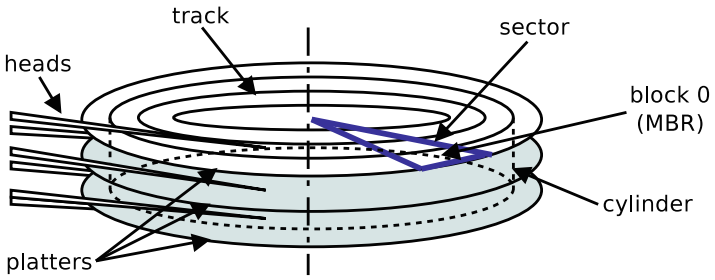


Fig. 3.6 A hard disk

3.3 Partitions

In computer systems, disk drives are, typically, used for secondary storage (though solid state devices are growing in popularity). Disk drives are read/write random access devices and comprise rigid platters rotating about a motor-driven spindle. The platters are coated with a thin layer of magnetic material. Read/write heads float very close the disk surfaces altering the alignment of magnet domains. The magnetic alignment of the domains is used to represent binary data. There are two methods of addressing data on hard disks:

- Logical block addressing (LBA)
- Cylinder-head-sector (CHS)

LBA supersedes the more complex CHS method which references blocks by a cylinder, head and sector three-tuple. Figure 3.6 shows a schematic of the hard disk. Note that each platter has two heads because it has a top and bottom surface. The CHS method of addressing reveals the physical details of disk. Furthermore, CHS only really applies to true hard drives, while solid state drives can report CHS values, they clearly do not reflect the actual geometry of the drive (as we will demonstrate when we create a virtual disk).

3.3.1 Partitions and Boot Sectors

At a logical level, the Linux kernel interacts with filesystems rather than a disk itself. Hard disks can be partitioned to accommodate multiple filesystems, in this way kernel sees each filesystem as a *logical* device that are referenced by a logical index. Device files (described above) exist in the filesystem for controlling block devices

(either the physical disk or the filesystems themselves). The command-line below shows the major and minor numbers for a hard disk and its partitions:

```
$ stat -c'%N %F %t %T' /dev/sda*
‘/dev/sda’ block special file 8 0
‘/dev/sda1’ block special file 8 1
‘/dev/sda2’ block special file 8 2
‘/dev/sda3’ block special file 8 3
‘/dev/sda5’ block special file 8 5
‘/dev/sda6’ block special file 8 6
```

This gives details of the system hard disk. The major number (in this case 8) references the device driver used to control device. The minor number (0–6 in this case) refers to the instance of the device. The */dev/sda* device, with the minor number 0, is the physical hard disk. The block devices with non-zero minor numbers (*sda1–sda6*) are for accessing the filesystems on the disk.

Information about the partitions (as well as code for bootstrapping the operating system) is kept in a *boot sector*. Boot sectors reside, either on the first block of the disk (MBR) or in first block of a partition (VBR). See Sect. 2.1 for a more detailed explanation of MBR and VBR.

For the purpose of this explanation, we will assume we are dealing with a partitioned disk and the boot sector is, therefore, an MBR. The system BIOS makes no distinction between MBRs and VBRs nor does it understand partitioning. The MBR may contain a bootstrap program *or* it may contain a boot manager that *chain loads* bootstraps code in a VBR (on one of the partitions). Table 3.2 describes the structure of the MBR.

Table 3.2 Master boot record sector structure

Description	Sector offset		Length
	Start	End	(bytes)
Code area	0	445	446
Master partition table	446	509	64
Boot record signature	510	511	2

Table 3.3 Primary partition table

Primary partition table entry	Sector offset	
	Start	End
First	446	461
Second	462	477
Third	478	493
Fourth	494	509

Table 3.4 Partition table entry

Field	Offset	Length	Description
Status	0	1	0x80 bootable
			0x00 non-bootable
Start address	1	3	In CHS form
Partition type	4	1	
End address	5	3	In CHS form
First block of partition	8	4	In LBA form
Partition size	12	4	

The partition table has space for four partition entries and, therefore, limits the number of (primary) partitions to four. Each entry is 16-bits in length, the offsets with the boot sector are shown in Table 3.3. Table 3.4 shows the structure of a partition table entry.

In the previous section, we used a regular file to create a *virtual* disk. Here, we use a file to create a virtual disk and demonstrate how the "disk" can be partitioned. Use the command-line below to create a 256 Mbyte file *disk.img*:

```
$ dd if=/dev/zero of=disk.img bs=1024 count=262144
262144+0 records in
262144+0 records out
268435456 bytes (268 MB) copied, 3.96965 s, 67.6 MB/s
```

The fdisk utility is an interactive partition table manipulator. Run fdisk on the virtual disk:

```
$ sudo fdisk -cuC 8 disk.img
Device contains neither a valid DOS partition table,
nor Sun, SGI or OSF disklabel
Building a new DOS disklabel with disk identifier
0x8f177a4d.
Changes will remain in memory only, until you decide
to write them.
After that, of course, the previous content won't be
recoverable.

Warning: invalid flag 0x0000 of partition table 4 will
be corrected by w(rite)

Command (m for help):
```

First verify that there are no partitions:

```
Command (m for help): p

Disk disk.img: 0 MB, 0 bytes
255 heads, 63 sectors/track, 8 cylinders, total 0
sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x8f177a4d

   Device Boot      Start         End      Blocks   Id  System
```

Create a primary partition with the `n` command:

```
Command (m for help): n
Command action
   e   extended
   p   primary partition (1-4)
p
Partition number (1-4): 1
First sector (1-128519, default 1):
Using default value 1
Last sector, +sectors or +size{K,M,G} (1-128519,
default 128519): 16M
```

Designate this partition type as a FAT32 filesystem (code b):

```
Command (m for help): t
Hex code (type L to list codes): b
Changed system type of partition 1 to b (W95 FAT32)
```

Mark this partition as the boot partition:

```
Command (m for help): a
Partition number (1-4): 1
```

Create another primary partition (64M in size):

```
Command (m for help): n
Command action
   e   extended
   p   primary partition (1-4)
```

```
p
Partition number (1-4): 2
First sector (17-128519, default 17):
Using default value 17
Last sector, +sectors or +size{K,M,G} (17-128519,
default 128519): 64M
```

By default, this partition will be Linux partition. View the newly created partitions:

```
Command (m for help): p

Disk disk.img: 0 MB, 0 bytes
255 heads, 63 sectors/track, 8 cylinders, total 0 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x8f177a4d

   Device Boot   Start    End    Blocks   Id  System
disk.img1   *        1     16         8    b  W95 FAT32
disk.img2           17     64        24   83  Linux
```

Write this partition information to the boot sector (MBR):

```
Command (m for help): w
The partition table has been altered!

WARNING: If you have created or modified any DOS 6.x
partitions, please see the fdisk manual page for additional
information.
Syncing disks.
```

Now examine the first block of *disk.img*. Initially *disk.img* contained all zeros, but after running fdisk we can see that the virtual disk has a disk identifier of 0x8f177a4d (note little endian is used):

```
$ od -x -N 16 -j432 -A d -w16 disk.img
0000432 0000 0000 0000 0000 7a4d 8f17 0000 0000
0000448
```

The boot record signature (0x55aa) has been written to the last two bytes of the block:

```
$ od -x -N 16 -j496 -A d -w16 disk.img
0000496 0000 0000 0000 0000 0000 0000 0000 aa55
```

Table 3.5 Partition entry

Partition	Offet	Fields					
		Status	CHS-start	Type	CHS-end	LBA-start	Size
First	446	80	00 02 00	0b	00 11 00	0000 0001	0000 0010
Second	462	00	00 07 01	83	01 02 02	0000 0011	0000 0030

The command-line below shows the area of the MBR for the partition table:

```
$ od -x -N 64 -A d -j446 -w16 disk.img
0000446 0080 0002 000b 0011 0001 0000 0010 0000
0000462 0000 0012 0183 0002 0011 0000 0030 0000
0000478 0000 0000 0000 0000 0000 0000 0000 0000
*
0000510
```

There are two entries, one at offset 446 and one at 462 corresponding to the two partitions we created with fdisk. To make things clearer we have present the partition records above in tabular form in Table 3.5. The status field (byte 0) of the first partition table entry is 0x80, indicating the partition in bootable. Conversely, the second partition is non-bootable because its status field is 0x00. We can also see type fields are 0x0b and 0x83 showing the partitions are for FAT32 and Linux (ext) filesystems, respectively.

We domonstrate how to format these two partitions. We cannot use the filesystem format utilities directly on *disk.img* like we did with *testfs*. Whereas, *testfs* was treated a filesystem and could be formatted, *disk.img* is emulating an entire disk. It is the partitions within it we wish to format not the file itself. In order to do this we must associate the file with a loopback device. We must also associate the partitions with loopback devices too. For this we use the losetup command. Associate the *disk.img* with a loopback device:

```
$ sudo losetup /dev/loop0 disk.img
```

We can see that */dev/loop0* has been associated with *disk.img*:

```
$ sudo losetup -a
/dev/loop0: [0805]:3944832 (/tmp/disk.img)
```

Also, running fdisk on */dev/loop0* shows partition table we created ealier:

```
$ sudo fdisk -lcu /dev/loop0

Disk /dev/loop0: 268 MB, 268435456 bytes
255 heads, 63 sectors/track, 32 cylinders, total 524288 sectors
```

```
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x8f177a4d

      Device Boot  Start    End  Blocks   Id  System
/dev/loop0p1           1     16       8    b  W95 FAT32
/dev/loop0p2          17     64       24   83  Linux
```

Next associate the first partition with loopback device */dev/loop1*:

```
$ sudo losetup -o 512 /dev/loop1 /dev/loop0
```

We need to tell losetup the start of the partition using the -o (offset) option. The results of fdisk (above) show us that the partition starts at block 1 (the second block) so the offset is 512 bytes. Format the loop1 partition as a MSDOS filesystem (we omit the output from the format command):

```
$ sudo mkdosfs /dev/loop1
```

The filesystem has not been created successfully in first partition of the disk. Create a mount point and mount the filesystem:

```
$ mkdir msdos
$ sudo mount -t msdos /dev/loop1 msdos
```

Display details of the filesystem with the stat command:

```
$ stat -f msdos
  File:"msdos"
    ID: 70100000000 Namelen: 12      Type: msdos
Block size: 4096       Fundamental block size: 4096
Blocks: Total: 65467      Free: 65467      Available: 65467
Inodes: Total: 0          Free: 0
```

We repeat the procedure above so that we can format the second partition. Calculate the offset of the partition. Given that fdisk tells us the second partition starts at block 64, the offset is $64 \times 512 = 32768$ bytes. Associate the partition with */dev/loop2*:

```
$ sudo losetup -o 32768 /dev/loop2 /dev/loop0
```

Format the loop2 partition as an ext2 filesystem (we omit the output of the command):

```
$ sudo mkfs.ext2 /dev/loop2
```

Create a mount point and mount the filesystem on it:

```
$ sudo mount /dev/loop2 mnt
```

Check the details of the filesystem:

```
$ stat -f mnt
  File: "mnt"
    ID: 96887f21ce887972 Namelen: 255     Type: ext2/ext3
Block size: 1024       Fundamental block size: 1024
Blocks: Total: 253839     Free: 251777     Available: 238672
Inodes: Total: 65536      Free: 65525
```

3.3.2 Extended Partitions

As we saw with fdisk, when we created a primary partition we had only a choice of four partition numbers. This is because the partition table can only have four entries. Extended partitions were introduced to overcome this limitation. One (and only one) partition in the partition table can be designated an extended partition.

```
$ sudo umount /dev/loop[12]
$ sudo losetup -d /dev/loop[0-2]
$ sudo fdisk -cu disk.img
```

Create a partition:

```
Command (m for help): n
Command action
   e   extended
   p   primary partition (1-4)
```

Select extended partition:

```
e
```

Choose the next available primary partition to be the extended partition (in this case 3):

```
Partition number (1-4): 3
```

Hit return at the next set of prompts to select default values:

```
First sector (65-128519, default 65):
```

```
Using default value 65
Last sector, +sectors or +size{K,M,G} (65-128519, default 128519):
Using default value 128519
```

Display the partition table:

```
Command (m for help): p

Disk disk.img: 0 MB, 0 bytes
255 heads, 63 sectors/track, 8 cylinders, total 0 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x8f177a4d

   Device Boot   Start      End   Blocks   Id  System
disk.img1   *        1       16        8    b  W95 FAT32
disk.img2           17       64       24   83  Linux
disk.img3           65   128519    64227+   5  Extended
```

Having set up an extended partition we need to add a partition for a filesystem:

```
Command (m for help): n
Command action
   l   logical (5 or over)
   p   primary partition (1-4)
```

Here we select a logical partition which is located within the extended partition:

```
l
```

Select default values by entering return at the following set of prompts:

```
First sector (66-128519, default 66):
Using default value 66
Last sector, +sectors or +size{K,M,G} (66-128519, default 128519):
Using default value 128519
```

Verify that the new partition has been created (as a partition for a Linux filesystem):

```
Command (m for help): p

Disk disk.img: 0 MB, 0 bytes
255 heads, 63 sectors/track, 8 cylinders, total 0
```

```
sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x8f177a4d

   Device Boot  Start      End   Blocks   Id  System
disk.img1   *       1       16        8    b  W95 FAT32
disk.img2          17       64       24   83  Linux
disk.img3          65   128519    64227+   5  Extended
disk.img5          66   128519    64227   83  Linux
```

Write the partition table to the MBR:

```
Command (m for help): w
The partition table has been altered!

Syncing disks.
```

We can see that a third partition table entry exists in the partition table (offset 478).

```
$ od -x -N 64 -A d -j446 -w16 disk.img
0000446 0080 0002 000b 0011 0001 0000 0010 0000
0000462 0000 0012 0183 0002 0011 0000 0030 0000
0000478 0100 0003 fe05 073f 0041 0000 f5c7 0001
0000494 0000 0000 0000 0000 0000 0000 0000 0000
0000510
```

The extended partition has its own partition table (in the bootblock). The offset of this partition tables is $65 \times 512 + 446 = 33726$ bytes. We can see that there is a partition record for the logical partition:

```
$ od -x -N 64 -A d -j33726 -w16 disk.img
0033726 0100 0004 fe83 073f 0001 0000 f5c6 0001
0033742 0000 0000 0000 0000 0000 0000 0000 0000
*
0033790
```

The partition record for the extended partition (third record in the MBR) and the EPBR are shown in Table 3.6.

Table 3.6 Partition entry

Partition	Offset	Fields					
		Status	CHS-start	Type	CHS-end	LBA-start	Size
Extended	478	00	01 03 00	05	fe 3f 07	0000 0041	0001 f5c7
EPBR	33726	00	01 04 00	83	fe 3f 07	0000 0001	0001 f5c6

3.4 Summary

The GNU/Linux file space (like that of Unix) comprises one or more *demountable volumes*, otherwise known as filesystems. A physical hard drive can be divided into a number of filesystems.

The default GNU/Linux filesystem is the extended filesystem (ext2, ext3 or ext4). GNU/Linux, however, supports many different filesystem types which is particularly important for embedded systems. VFS (virtual filesystem) implements a layer of abstraction in order to provide a common interface to physical filesystems, regardless of their type.

We described in detail the anatomy of a hard disk by using a regular file as a virtual disk. We showed how to partition the (virtual) disk so that it could hold multiple filesystems. This enabled us to carry out a detailed analysis of the disk in order to understand its structure.

Furthermore, the tools we used to partition the disk will come in useful for subsequent chapters when we build an embedded system.

Reference

1. Haviland K, Salama B (1987) Unix system programming. Addison-Weseley, Boston

Building an Embedded System (First Pass)

4

In this chapter and the next (Chap. 5) we show how to build embedded GNU/Linux systems. This chapter is a *first pass*. We use various tools to generate the software components so that we can focus on the structure of the system rather than the details. In Chap. 5 (second pass), we cover the topic in greater depth construct a system from the source code. An operating system (embedded or otherwise) comprises three primary components:

- Bootloader
- Kernel
- Root filesystem.

The filesystem that is mounted on the root directory of the file space is the *root filesystem*. Files required by the operating system in order to function are stored within this directory hierarchy, these include:

- Configuration files
- Start-up/shutdown scripts
- Shared libraries
- System utilities
- User applications.

We use the *debootstrap* utility to create the root directory hierarchy. The debootstrap tool creates a base Debian directory hierarchy with (nearly) all the utilities, libraries and configuration files required to run the operating system. The only components that are not generated by debootstrap are the kernel and bootloader.

The Linux kernel itself is stored within the directory hierarchy, though not necessarily on the root filesystem. Typically, the kernel (along with bootloader configuration files) are stored under the */boot* directory. Sometimes a separate filesystem is mounted on */boot*, otherwise */boot* is merely a directory on the root filesystem.

Instead of running the operating system natively on a physical machine we run it under the control of a virtual machine on the host system upon which it was built.

A. Holt and C.-Y. Huang, *Embedded Operating Systems*, Undergraduate Topics in Computer Science, DOI 10.1007/978-1-4471-6603-0_4

User mode Linux (UML) is a virtualization technology whereby the Linux kernel is compiled to run as a ELF (executable and linkable format) binary which can be run as a normal user process on a host GNU/Linux system. We use UML to simplify the process of building a kernel. This also obviates the need for a bootloader.

The aim of this chapter is to build a fully functional operating system. We will run it as a virtual machine on a GNU/Linux host. While it will not run natively on an embedded platform, it could with a little extra work. While both of the systems built in this chapter and Chap. 5 are fully functional, they are mainly for educational purposes to illustrate the concepts of embedded operating systems. We do not recommend you use them as production systems.

4.1 Creating the Root Filesystem

We use the debootstrap tool to generate a base Debian system. The debootstrap tool downloads *.deb* packages from on-line repositories and unpacks them to form a complete system. These repositories are on-line so host machine will need Internet access. The debootstrap utility can be installed using the package management system:

```
$ sudo apt-get install debootstrap
```

We create the root directory structure for our embedded system in a sub-directory on our host system. The command-line below creates sub-directories *deb* and *deb/root*:

```
$ mkdir deb/{,root}
```

We will use *deb* as a directory to work under. The *deb/root* directory is the root directory ("/") for the embedded system. Set environment variables to the path names of these directories:

```
$ export BASE=${HOME}/deb
$ export ROOT=${BASE}/root
```

In the example above we created the *deb* directory under our home directory ($HOME). If you have created *deb* in a different part of the directory hierarchy, then set BASE and ROOT environment variables to reflect the directory path names accordingly. Change the current directory to $BASE:

```
$ cd $BASE
```

When we run debootstrap it will create a default base system. We can specify additional packages to be included in that base system (and, conversely, packages to be excluded). We define the packages we wish to include in an environment variable:

```
$ export IPKG="udev,locales,resolvconf,rcconf,ssh"
```

Command-line arguments to be passed to debootstrap are defined in an environment variable array:

```
$ OPTS[0]="--arch=i386"
$ OPTS[1]="--include=${IPKG}"
```

These options inform debootstrap of the processor architecture (in this case, Intel 386) and the packages to include: The command below creates a Debian Squeeze distribution in the $ROOT directory:

```
$ sudo debootstrap ${OPTS[*]} squeeze ${ROOT}
```

The command above may take some time to run depending upon your system and the speed of your Internet connection. Once debootstrap has completed, we can see that a number of sub-directories have been created under $ROOT:

```
$ ls ${ROOT}
bin   dev  home  media  opt   root  selinux  sys  usr
boot  etc  lib   mnt    proc  sbin  srv      tmp  var
```

While debootstrap has created all the necessary configuration files, it is necessary to customise some of them. Ensure that the ROOT environment variable is set correctly with the path name of the "root" of the embedded system we are creating. If $ROOT is unset then you could be modifying the configuration files on you host by mistake. Edit *$ROOT/etc/fstab* and add the content shown in Listing 4.1.

Listing 4.1 *Contents of* /etc/fstab

```
proc /proc proc rw,nodev,nosuid,noexec 0 0
tmpfs /dev/shm tmpfs defaults 0 0
tmpfs /tmp tmpfs defaults 0 0
```

The *fstab* file contains descriptive information about the various file systems. The first line of *fstab* mounts a pseudo filesystem on */proc*. The proc filesystem is not a filesystem in the traditional sense, rather it is an interface into the runtime state of the kernel and running processes.

The next two lines of *fstab* specify temporary filesystems mounted on */dev/shm* and */tmp*. The */tmp* directory is merely for temporary files (and temporary subdirectories). Any files created in */tmp* will not persist after a reboot. */dev/shm* is a RAM disk implementation of shared memory.

The name of the system is stored in *$ROOT/etc/hostname*. Edit the file so that it contains the desired system name. The choice of system name is arbitrary, in our case we call the system "deb". Listing 4.2 shows an example content for *hostname* file.

Listing 4.2 *Contents of* /etc/hostname

```
deb
```

The init process is created by the Linux kernel on boot up. It is responsible for starting other processes on the system. The configuration file for init process is */etc/inittab*. It instructs the init process to start new processes. For example, the line below spawns a getty process (Get TTY) on a TTY port (in this case, tty1):

```
1:2345:respawn:/sbin/getty 38400 tty1
```

In addition, the lines below will cause init to spawn gettys on the two serial lines ttyS0 and ttyS1. These lines will probably exist in the inittab but will be commented out. Uncomment the lines to activate gettys on the serial interfaces:

```
T0:23:respawn:/sbin/getty -L ttyS0 19200 vt100
T1:23:respawn:/sbin/getty -L ttyS1 19200 vt100
```

We will look at Ethernet network configuration later in this chapter. For now, we just configure the loopback interface lo, Listing 4.3 shows the content of *$ROOT/etc/network/interfaces*.

Listing 4.3 *Lines commented out in* /etc/network/interfaces

```
auto lo
iface lo inet loopback
```

At this point, it is sensible to test the system so far. The chroot command allows us to change the apparent root directory of the host system:

```
$ sudo chroot $ROOT
root@deb:/#
```

When we list the content of the current directory, we are actually viewing the content of *$HOME/deb/root*:

```
# ls
bin   dev  home  media  opt   root  selinux  sys  usr
boot  etc  lib   mnt proc  sbin  srv      tmp  var
```

However, if we check the pathname of the current directory, pwd returns "/" rather than the absolute pathname of the directory from the host's root:

```
# pwd
/
```

Furthermore, we cannot access directories above *$HOME/deb/root*. We are, therefore, not using commands or libraries of the host system but of the embedded system itself. This serves as an initial test of the embedded system, verifying that the software and any dependencies are functioning correctly. As we running under the chroot jail, we can take the opportunity to set the root password:

```
# passwd
Enter new UNIX password: letmein
Retype new UNIX password: letmein
passwd: password updated successfully
```

Note that, the password program does not echo the password to the screen, thus you will not see the string "letmein". Furthermore, we recommend you use stronger passwords than the examples given in this book.

The locales package needs to be reconfigured. This is done by running the command:

```
# dpkg-reconfigure locales
```

Select the appropriate locales from the first menu. In our example, we select:

```
en_GB ISO-8859-1
en_GB.ISO-8859-15 ISO-8859-15
en_GB.UTF-8 UTF-8
```

From the next menu, select:

```
en_GB
```

To exit from the chroot jail, simply issue the command:

```
# exit
```

4.2 Building a Linux Kernel

In this section we describe how to build a UML (user mode Linux) kernel. The procedure for building a UML kernel is virtually the same as building a kernel that runs natively. Download the Linux kernel source file:

```
$ cd $BASE  # ensure we are in the build directory
$ wget ftp://ftp.kernel.org/pub/linux/kernel/v2.6/\
> linux-2.6.39.4.tar.gz
```

Extract the kernel sources files from the tar file:

```
$ tar zxvf linux-2.6.39.4.tar.gz
```

Change the directory to the Linux kernel source:

```
$ cd linux-2.6.39.4
```

The Linux kernel is highly configurable. The features which are compiled in the final kernel image are determined by a plethora of options. The command below runs a menu driven configuration tool:

```
$ make menuconfig ARCH=um
```

Figure 4.1 shows the top level menu of configuration tool. The option ARCH=um in the command-line above specifies the compilation of a UML kernel and sets all the necessary options. We can immediately select "exit" and save the configuration. Compile the kernel with the command-line below:

```
$ make linux ARCH=um
```

The kernel can be compiled with *loadable kernel modules*. Closer examination of Fig. 4.1 shows that, the option "Enable loadable module support", is set. This means that some of the features of the kernel will not be built into the kernel image but built as separate modules (in separate files). To compile the modules, issue the command-line below:

```
$ make modules ARCH=um
```

As we are building a UML kernel, we do not need to install the kernel image on the system's root filesystem. However, we do need to install the kernel modules on the filesystem. Check the ROOT environment variable is set correctly before issuing the command below:

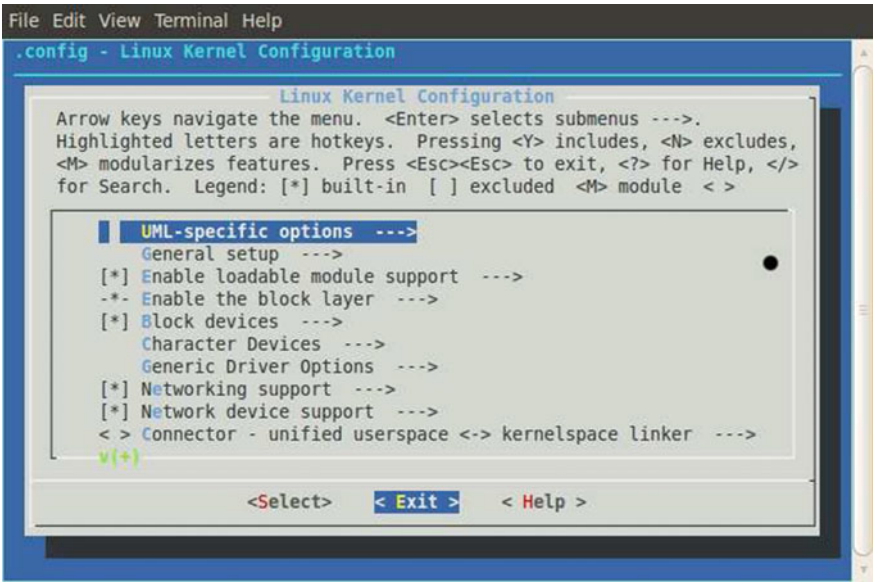


Fig. 4.1 Makeconfig UML

```
$ sudo make modules_install INSTALL_MOD_PATH=$ROOT \
> ARCH=um
```

View the modules installed under the system's root directory:

```
$ tree -L 2 $ROOT/lib/modules
/home/aholt/emx/uml/root/lib/modules
`-- 2.6.39.4
    |-- build -> /home/aholt/deb/linux-2.6.39.4
    |-- kernel
    |-- modules.alias
    |-- modules.alias.bin
    |-- modules.builtin
    |-- modules.builtin.bin
    |-- modules.ccwmap
    |-- modules.dep
    |-- modules.dep.bin
    |-- modules.ieee1394map
    |-- modules.inputmap
    |-- modules.isapnpmap
    |-- modules.ofmap
    |-- modules.order
    |-- modules.pcimap
```

```
      |-- modules.seriomap
      |-- modules.symbols
      |-- modules.symbols.bin
      |-- modules.usbmap
      `-- source -> /home/aholt/deb/linux-2.6.39.4
4 directories, 17 files
```

4.3 Build the Root Filesystem

As we are running our embedded system in a virtual machine, we can create the root filesystem in a *virtual partition*. We create this partition in a regular file which we format as a filesystem. Make the build directory the current directory:

```
$ cd $BASE
```

The command-line below creates a 512M (regular) file:

```
$ dd if=/dev/zero of=deb_fs seek=512 count=1 bs=1M
1+0 records in
1+0 records out
```

We format the (*deb_fs*) file as an ext2 filesystem (just as we would for a physical block device):

```
$ mkfs.ext2 -F deb_fs
```

The command will produce many lines of output but, for brevity, we omit them here. We need to transfer the root directory structure (rooted at $ROOT) onto the filesystem (*deb_fs*). Create a tar (tape archive) file of the $ROOT directory:

```
$ sudo tar zcvf deb_root.tar.gz $ROOT
```

Mount *deb_fs* as a loop device on the $ROOT directory:

```
$ sudo mount -o loop deb_fs $ROOT
```

Untar the *deb_root.tar.gz* and unmount the loop device:

```
$ sudo tar zxvf deb_root.tar.gz
$ sudo umount $ROOT
```

The filesystem in *deb_fs* now contains the root directory structure.

4.4 Running UML

When we compiled the UML kernel, it created an executable file *linux* in kernel source directory. For convenience, we create a symbolic link in the $BASE directory to *$BASE/linux-2.6.39.4/linux*:

```
$ cd $BASE    # ensure we are in the build directory
$ ln -s linux-2.6.39.4/linux linux
```

The script in Listing 4.4 is a launch script for the UML process. Create it in the $BASE directory.

Listing 4.4 *UML Launch Script: uml.sh*

```
#!/bin/bash

FS="deb_fs"
UMID="${FS%%_*}"

OPTS[0]="ubd0=$FS"
OPTS[3]="con=pts con0=fd:0,fd:1"
OPTS[4]="ssl=pts"

while getopts ": n : s : r :" CMD_LINE_OPTIONS
do
    case $CMD_LINE_OPTIONS in
        r     ) OPTS[0]="ubd0=$OPTARG"
                UMID="${OPTARG%%_*}"
              ;;
        s     ) OPTS[1]="ubd1=$OPTARG";;
        n     ) OPTS[2]="eth0=tuntap,$OPTARG,,";;
        *     ) echo "Unimplemented option";;
    esac
done

OPTS[5]="umid=${UMID}"

./linux ${OPTS[*]}
```

Make the launch script executable:

```
$ chmod +x uml.sh
```

We can now start our UML system simply by running the script:

```
$ ./uml.sh
```

A sequence of kernel debug messages are displayed on the screen as the system boots up. When the init process starts it runs gettys on the consoles and serial lines. At the end of the boot process you should see some lines like those below:

```
Serial line 0 assigned device '/dev/pts/3'
Serial line 1 assigned device '/dev/pts/4'
Virtual console 6 assigned device '/dev/pts/5'
Virtual console 5 assigned device '/dev/pts/6'
Virtual console 4 assigned device '/dev/pts/7'
Virtual console 3 assigned device '/dev/pts/8'
Virtual console 2 assigned device '/dev/pts/9'
Virtual console 1 assigned device '/dev/pts/10'
```

It shows us which pseudo terminals on the host are attached to the consoles and serial lines of the host. We can login to console 1 with:

```
$ screen /dev/pts/10
```

We are presented with a login prompt (it may be necessary to hit a key to wake up the getty). Login as "root" and use the password we set earlier:

```
Debian GNU/Linux 6.0 deb tty1

deb login: root
Password: letmein
```

Note the string "letmein" is not echoed to the screen. The message of the day is displayed followed by a command-line prompt:

```
root@deb:~#
```

Minicom can be used as an alternative to screen. Use the command-line below to login through serial 0 (ttyS0):

```
$ minicom -o -p /dev/pts/3
```

Login in as root and issue the following command to confirm your tty:

```
root@deb:~# tty
/dev/ttyS0
```

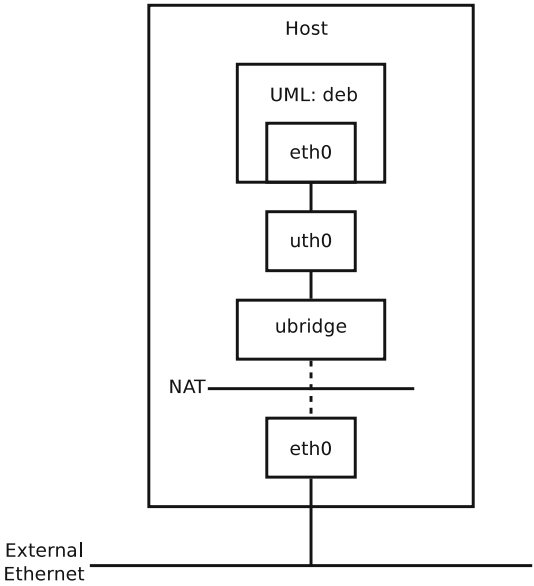


Fig. 4.2 UML networking architecture

4.5 Networking

In this section we describe how to configure the deb virtual machine (VM) for networking. Figure 4.2 shows the host system with the UML deb system running as a virtual machine. The host has an Ethernet interface eth0. This is a physical interface card connected to an external Ethernet network (which in turn is connected to the wider Internet). In our case, the IP details of eth0 were configured through the dynamic host control protocol (DHCP).

The deb virtual machine also has an eth0 interface which is bound to a virtual network interface uth0 on the host. In addition, a virtual bridge (ubridge) is configured on the host and uth0 is attached to it. Figure 4.2 also shows the network address translation (NAT) between the bridge and the host's eth0 interface. The reason for this will be explained later.

The eth0 interface on the deb VM has to be configured with IP details (address, network mask, default gateway etc.). We need to configure the *interfaces* file on the deb filesystem. If the UML VM is running, shut it down from its command-line:

```
root@deb:~# shutdown -h now
```

Mount *deb_fs* as a loop device on the $ROOT directory.

```
$ cd $BASE
$ sudo mount -o loop deb_fs $ROOT
```

Edit the *$ROOT/etc/network/interfaces* file and add the IP details for eth0. Listing 4.5 shows the lines that need to be added:

Listing 4.5 *Configuration of eth0 (/etc/network/interfaces)*

```
auto eth0
iface eth0 inet static
    address 192.168.7.2
    netmask 255.255.255.0
    network 192.168.7.0
    broadcast 192.168.7.255
    gateway 192.168.7.1
    # dns-* options are implemented by the resolvconf
    # package, if installed
    dns-nameservers 8.8.8.8 8.8.4.4
    dns-search emx.local
```

Note, we use the Google DNS servers (8.8.8.8 and 8.8.4.4). While this will work, we recommend that you substitute your own local DNS servers. The choice of IP network is somewhat arbitrary but ensure it does not conflict with the network of the host's eth0 interface. It is highly likely that the network you choose will be unknown to any routers on the external Ethernet network and will be unable to forward packets to the deb VM. We can resolve this issue by adding NAT rules to the host. As we have completed all the necessary configuration of the deb VM, we can unmount *deb_fs*:

```
$ sudo mount $ROOT
```

All the configuration steps hereon in are performed on the host. Change the group and permissions of */dev/net/tun*

```
$ sudo chgrp aholt /dev/net/tun
$ sudo chmod 660 /dev/net/tun
```

Replace the group "aholt" with your group name (use the id command to reveal your group). Add a virtual bridge, ubridge:

```
$ sudo brctl addbr ubridge
```

Disable the spanning tree protocol on the bridge:

```
$ sudo brctl stp ubridge off
```

Give the bridge an IP address. This should match the gateway address in *$ROOT/etc/network/interfaces*:

```
$ ifconfig ubridge 192.168.7.1 netmask 255.255.255.0 up
```

Create a virtual Ethernet interface and bring it up:

```
$ sudo tunctl -u aholt -t uth0
$ sudo ifconfig uth0 0.0.0.0 promisc up
```

Attach interface uth0 to bridge ubridge:

```
$ btctl addif ubridge uth0
```

We can verify the bridge and virtual interface thus:

```
$ brctl show ubridge
bridge name bridge id STP enabled interfaces
ubridge 8000.9a48ef8bbcbd no uth0
```

We are now in a position to launch a deb VM with networking capabilities:

```
$ ./uml.sh -n uth0
```

Once the system has booted we can test the network functionality from the host with the ping utility:

```
$ ping -c 4 192.168.7.2
```

Conversely, we should be able to ping the host (on 192.168.7.1) from the UML VM. The VM cannot, however, communicate with any devices beyond the host. We will resolve this problem now. The host must be configured to forward IP packets (just like a router). We do this through the proc filesystem interface. Enable forwarding by writing the value 1 to */proc/sys/net/ipv4/ip_forward*:

```
$ sudo -i
# echo 1 > /proc/sys/net/ipv4/ip_forward
# exit
logout
```

Notice that in order to write values to the *ip_forward* file can only be done as root and *not* through sudo. As we mentioned above, the routers on the external Ethernet network will have no knowledge of the 192.168.7.0/24 network and will, therefore, be unable to forward packets to the VM. We will use NAT (network address translation) to circumvent this problem. The Linux kernel includes a network packet filtering subsystem called Netfilter [1]. The primary purpose of Netfilter is to support firewall functionality within the Linux Kernel, however, it has other uses, including NAT. The iptables command is used to manage and configure Netfilter. Using iptables,

sequences of packet processing rules are configured. First delete any existing iptables rules:

```
$ iptables -F
```

Always accept loopback traffic:

```
$ iptables -A INPUT -i lo -j ACCEPT
```

Configure NAT (masquerading):

```
$ iptables -A POSTROUTING -t nat -o eth0 \
> -s 192.168.7.0/24 -d 0/0  -j MASQUERADE
```

Configure forwarding rules:

```
$ iptables -A FORWARD -t filter -o eth0 -m state \
> --state NEW,ESTABLISHED,RELATED -j ACCEPT
$ iptables -A FORWARD -t filter -i eth0 -m state \
> --state ESTABLISHED,RELATED -j ACCEPT
```

Shut down the deb VM and restart it (with networking enabled). Login to the VM remotely from the host system. Verify the VM has connectivity to the Internet:

```
root@deb:~# ping -c 4 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_req=1 ttl=45 time=25.1 ms
64 bytes from 8.8.8.8: icmp_req=2 ttl=45 time=24.6 ms
64 bytes from 8.8.8.8: icmp_req=3 ttl=45 time=36.8 ms
64 bytes from 8.8.8.8: icmp_req=4 ttl=45 time=24.8 ms
--- 8.8.8.8 ping statistics ---
4 packets transmitted, 4 received, 0
time 3011ms
rtt min/avg/max/mdev = 24.627/27.871/36.866/5.200 ms
```

4.6 Summary

In this chapter we described how to build an embedded system using the debootstrap utility and a UML kernel. With debootstrap, we built the root filesystem with all the shared libraries, utilities, start-up scripts and configuration files required by the operating system. We used a UML kernel so that we could run the system as a virtual machine (VM) on a host GNU/Linux system. The UML kernel obviates the need

for a bootloader because we can start the kernel like any other user process. We also showed how the system could be networked and access the Internet just like a system running natively.

Reference

1. Purdy GN (2004) Linux iptables—pocket reference: firewalls, NAT and accounting. O'Reilly, USA

Building an Embedded System (Second Pass)

5

In the previous chapter we showed how to build an embedded system which ran in a virtual machine (VM). We will refer to this system as the "deb" system to distinguish it from the embedded system described in this chapter.

The root filesystem for the deb system was generated using the debootstrap utility. This created the root directory structure and installed the necessary libraries, binary utilities, scripts and configuration files required for the system. To complete the root filesystem, we only needed to make minor amendments to some of the configuration files. We built a UML kernel so that we could run deb as a VM on a host machine. This obviated the need for a bootloader. Debootstrap and UML greatly simplified the task of building the system.

In this chapter, however, we want to examine the components of an embedded GNU/Linux system in more detail. We show how to build an system without the aid of automated tools such debootstrap. Further, we will build a system that runs natively of an embedded platform. It is minimal system that we build, but it is fully functional. We divided the build process into several stages which are outlined below:

Glibc Much of the software of a Unix (and Unix-like) system is written in the C langauge. Common operations, such as input/output, memory management, string manipulation, for example, are not part of the language itself. Rather, they are provided through a standard library (either statically linked in at compile time or dynamically at run time). Glibc is a standard C library for GNU/Linux.

Ncurses Ncurses provides a library of screen-handling functions for writng GUI-like applications in text-based terminals. Ncurses functions form a wrapper around terminal control-codes so that applications can be developed in a terminal independent way.

Busybox Most of the utilities in the deb system were from part of the coreutils and util-linux packages. These packages are designed for general purpose operating systems. For systems with limited resources (such as embedded systems) coreutils and util-linuux can take up a lot of disk space. The Busybox package provides common GNU/Linux utilities but is designed specifically for embedded

A. Holt and C.-Y. Huang, *Embedded Operating Systems*, Undergraduate Topics in Computer Science, DOI 10.1007/978-1-4471-6603-0_5

environments. Busybox is highly customisable. enabling cut down versions of the original Unix utilities to be compiled into a single binary image.
Bash Bourne again shell (Bash) is a Unix command interpreter. While Busybox can provide a number of shell utilities we prefer a full version of Bash.
Sysvinit The kernel starts the init process at boot-time. The init process, in turn, starts all the services (daemon processes) on the system as well as *getty* processes which wait for logins on console and tty devices. We compile the init utility from the Sysvinit package.
The kernel In the previous chapter we compiled a UML kernel, that is, a kernel that can be run as a regular process. In this chapter we shall build a system to run on an actual embedded platform. Therefore, we need to build and install a kernel that runs natively.
The bootloader The bootloader is responsible for loading the kernel into memory and passing control to it. We use Extlinux which is a derivative of Syslinux. Unlike Syslinux, however, which can only boot a kernel from a FAT filesystem, Extlinux can boot the system from an ext filesystem.

In addition to building the software components of the system, there are also a number of administrative tasks that need to be carried out:

- Create configuration files.
- Create Start-up/Shutdown Scripts.
- Create special files for Unix devices.

We built this operating system to run on a Soekris net4521 [1]. The net4521 has an i486 133 MHz processor, 64 Mbyte SDRAM, two Ethernet ports, one Serial port and compact flash (CF) socket (see Fig. 5.1). It should be possible to run this system on any

Fig. 5.1 Soekris net4521

Intel based platform. Only the kernel compilation procedure requires special attention as the configuration options would need to be congruent to the target platform's hardware.

5.1 Preliminaries

There are a few administrative tasks we have to complete on the host machine before we start building the operating system. Install the build-essentials package using the distribution's package management:

```
$ sudo apt-get install build-essential
```

The following packages also need to be installed:

```
$ sudo apt-get install autoconf bison gawk nasm \
> dpkg-dev libncurses5-dev
```

As in the previous chapter, we create a directory (and several subdirectories) in which to build the system:

```
$ mkdir emx/{,src,root,src/build,src/tools}
```

Set environment variables to reflect the pathname of these directories:

```
$ export BASE=${HOME}/emx
$ export SRC=${BASE}/src
$ export ROOT=${BASE}/root
```

We recommend you periodically check that these environment variables are set, particularly when you install software. If they are not set you could overwrite files on your host system.

5.2 Glibc

In this section we show how to build the GNU C library, Glibc. Glibc is by far the most difficult component of system to build. Once you have successfully compiled Glibc, the other components are relatively straight forward. When building Glibc for your embedded system, we recommend you use same version of the library that is running on your host system. To find out the version of Glibc running on your host system, run:

```
$ /lib/libc.so.6
GNU C Library (Debian EGLIBC 2.11.2-10) stable release
version 2.11.2, by Roland McGrath et al.
Copyright (C) 2009 Free Software Foundation, Inc.
This is free software; see the source for copying
conditions.
There is NO warranty; not even for MERCHANTABILITY or
FITNESS FOR A
PARTICULAR PURPOSE.
Compiled by GNU CC version 4.4.5.
Compiled on a Linux 2.6.32 system on 2011-01-23.
Available extensions:
crypt add-on version 2.1 by Michael Glad and
others
GNU Libidn by Simon Josefsson
Native POSIX Threads Library by Ulrich Drepper
et al.
BIND-8.2.3-T5B
For bug reporting instructions, please see:
<http://www.debian.org/Bugs/>.
```

It can be seen from the first line of the output that the version is 2.11.2-10. It also shows the version of GCC used to compile the library (4.4.5) and the kernel version used for its headers (2.6.32). Download the compressed tar file for GLIBC:

```
$ wget http://ftp.gnu.org/pub/gnu/glibc/\
> glibc-2.11.2.tar.gz
```

Glibc requires the headers files from Linux kernel. We downloaded version 2.6.39.4 of the kernel in Sect. 4.2. Make *$SRC* the current directory and extract the source files from the tar file:

```
$ cd $SRC
$ tar zxvf ~/deb/linux-2.6.39.4.tar.gz
```

We do not need to compile the whole kernel at this stage, we just need to build the headers for Glibc:

```
$ cd linux-2.6.39.4
$ make headers_check
```

Install the headers in the *$SRC/tools* directory:

```
$ make INSTALL_HDR_PATH=${SRC}/tools headers_install
```

Now we can return to building Glibc itself. Extract the source files:

```
$ cd $SRC
$ tar zxvf glibc-2.11.2.tar.gz
```

Set the CFLAGS environment variable:

```
$ export CFLAGS="-march=i486 -O3 -fno-stack-protector"
```

Glibc, like many of the components we compile, requires many configuration options. For convenience and brevity we use an environment variable array to define them:

```
$ OPTS[0]="--prefix=/usr"
$ OPTS[1]="--with-headers=${SRC}/tools/include"
$ OPTS[2]="--host=i486-pc-linux-gnu"
$ OPTS[3]="--disable-profile"
$ OPTS[4]="--disable-sanity-checks"
$ OPTS[5]="--without-gd"
$ OPTS[6]="--without-selinux"
```

It is important to note Glibc cannot be built in its own source directory. We therefore use a separate *build* directory (which we created above) for this purpose:

```
$ cd $SRC/build
```

Configure Glibc with the command-line below:

```
$ ../glibc-2.11.2/configure ${OPTS[*]}
```

Then compile the software:

```
$ make
```

We complete the build process by installing Glibc under the $ROOT directory:

```
$ make install install_root=$ROOT
```

Create the *ld.so.conf* file in the *$ROOT/etc* with the contents shown in Listing 5.1.

Listing 5.1 *Contents of /etc/ld.so.conf*

```
/lib
/usr/lib
/usr/local/lib
```

5.3 Optimisation

If disk space is at a premium, we can reduce the size of some files by removing the symbols from object files. We can even remove some files altogether. Use the strip utility to remove the symbols from object files:

```
$ strip $ROOT/lib/*.so
```

Remove the following directories:

```
$ rm -rf $ROOT/usr/include
```

Use the sequence of commands below to remove redundant files:

```
$ cd $ROOT/usr/share/locale
$ ls | grep -v locale.alias | xargs rm -rf
$ cd $ROOT/usr/share/i18n/charmaps
$ ls  | grep -v -E  -w '8859|UTF' | xargs rm
$ cd $ROOT/usr/share/i18n/locales
$ ls  | grep -v -E -w 'en_GB|en_US|POSIX' | xargs rm
```

5.4 Ncurses

Ncurses is a library of screen-handling functions. Applications like vi make extensive use of Ncurses. Download the source code for Ncurses:

```
$ cd $SRC
$ wget http://ftp.gnu.org/pub/gnu/ncurses/\
> ncurses-5.4.tar.gz
```

Extract the source files:

```
$ tar zxvf ncurses-5.4.tar.gz
```

Make the Ncurses directory the working directory:

```
$ cd ncurses-5.4
```

Set the CFLAGS environment variable:

```
$ export CFLAGS="-march=i486 -O3"
```

We use the OPTS array to define the configuration options for Ncurses. As this array was used for Glibc above, we need to clear it:

```
$ unset OPTS
```

Set the configuration options:

```
$ OPTS[0]="--prefix=$ROOT"
$ OPTS[1]="--disable-database"
$ OPTS[2]="--with-fallbacks=linux,vt100,xterm,rxvt"
$ OPTS[3]="--with-shared"
$ OPTS[4]="--without-normal"
$ OPTS[5]="--without-debug"
$ OPTS[6]="--without-ada"
$ OPTS[7]="--without-progs"
$ OPTS[8]="--without-cxx"
$ OPTS[9]="--libdir=$ROOT/lib"
```

Configure Ncurses:

```
$ ./configure ${OPTS[*]}
```

Compile and install the Ncurses library:

```
$ make && make install
```

We remove the symbols from the library file:

```
$ cd $ROOT
$ strip lib/libncurses.so.5.4
```

Finally we can delete the manual pages and include files:

```
$ sudo rm -rf man include
```

5.5 Busybox

Busybox provides many of the Unix commands and system utilities. Download the Busybox source files into the *$SRC* directory:

```
$ cd $SRC
$ wget http://busybox.net/downloads/\
```

```
> busybox-1.9.3.tar.gz
```

Extract the Busybox source from the tar file:

```
$ tar jxvf busybox-1.19.3.tar.bz2
```

Configure Busybox by running the command-line below:

```
$ cd busybox-1.19.3
$ make menuconfig
```

This runs a menu based configuration utility similar to that of kernel. We need to make one small modification to the default Busybox configuration. Busybox provides its own version of init. However, we found that there was a problem with the Busybox version of init running on the Soekris. For this reason we used the standard init from the Sysvinit package. We, therefore, need to configure Busybox to omit compiling init. Go to the Init Utilities section and disable the init option:

```
[ ] init
```

Now exit and save the configuration.

```
$ export CFLAGS="-march=i486"
$ export LDFLAGS="-L${ROOT}/lib"
```

Compile and install Busybox:

```
$ make && make CONFIG_PREFIX=$ROOT install
```

5.6 Bash

Bash is a command-line interpreter. This is the user interface for the system. Download the tar file:

```
$ wget http://ftp.gnu.org/gnu/bash/bash-4.2.tar.gz
```

Extract the sources files from the tar file:

```
$ tar zxvf bash-4.2.tar.gz
```

Set the following compiler flags:

```
$ export CFLAGS="-march=i486"
$ export LDFLAGS="-L${ROOT}/lib"
```

Set the configuration options:

```
$ unset OPTS
$ OPTS[0]="--prefix=$ROOT"
$ OPTS[1]="--with-curses"
$ OPTS[2]="--oldincludedir=${SRC}/ncurses-5.7/include"
$ OPTS[3]="--libdir=$ROOT/lib"
```

Configure Bash then compile it:

```
$ cd bash-4.2
$ ./configure ${OPTS[*]}
$ make
```

Finally install the Bash executable in bin directory:

```
$ cp bash $ROOT/bin
```

Create a symbolic link sh to bash:

```
$ cd $ROOT/bin
$ ln -s bash sh
```

5.7 Sysvinit

We install the init utility from Sysvinit package rather than using the Busybox version. Ensure you have disabled the init configuration option in the Busybox section above. For convenience, we use the package manager to download the Sysvinit source:

```
$ cd $SRC
$ apt-get source sysvinit
```

Set compiler flags (if not already set):

```
$ export CFLAGS="-march=i486"
```

There is no need for a configuration process, we can simply compile it:

```
$ cd $SRC/sysvinit-2.86.ds1/src
$ make
```

Install:

```
$ cp init $ROOT/sbin
```

The telinit utility is used to instruct init to perform certain actions. We "create" telinit simply by creating a symbolic link to init:

```
$ cd $ROOT/sbin
$ ln -s init telinit
```

5.8 Devices (/dev)

Special devices files are located in the */dev* directory. The deb system built in the previous chapter included a utility called udev which created the devices at boot-time. Here we need to create the device files manually at install time. First create the *dev* directory and some sub-directories under it:

```
$ cd $ROOT
$ mkdir dev/{,misc,shm,ubd}
```

We need to create a number character devices in *$ROOT/dev*. Create the console using mknod:

```
$ cd $ROOT/dev
$ sudo mknod -m 660 console c 5 1
```

Create the serial lines:

```
$ sudo mknod -m 660 ttyS0 c 4 64
$ sudo mknod -m 660 ttyS1 c 4 65
```

Create a number of tty devices:

```
$ sudo mknod -m 660 tty0 c 4 0
$ sudo mknod -m 660 tty1 c 4 1
$ sudo mknod -m 660 tty2 c 4 2
$ sudo mknod -m 660 tty3 c 4 3
$ sudo mknod -m 660 tty4 c 4 4
```

```
$ sudo mknod -m 660 tty5 c 4 5
$ sudo mknod -m 660 tty6 c 4 6
```

The ptmx device is used to create a master and slave pair for a pseudo terminal. Two file descriptors are returned to any process that open */dev/ptmx*. One descriptor is for a master pseudo terminal and the other is for the slave. Create the ptmx device:

```
$ sudo mknod ptmx c 5 2
```

Create real time clock device:

```
$ sudo mknod -m 660 misc/rtc c 10 135
```

The null device acts as a data sink for write operations and zero returns 0 characters when read. Create null and zero devices:

```
$ sudo mknod null c 1 3
$ sudo mknod zero c 1 5
```

While null and zero are treated as "devices" they do not correspond to actual physical hardware. Create the block devices for the hard disk, which in this case, is a compact flash card:

```
$ mknod --mode=660 hda b 3 0
$ mknod --mode=660 hda1 b 3 1
$ mknod --mode=660 hda2 b 3 2
```

The *hda* device is the disk itself, while the *hda*1 and *hda*2 devices are partitions of the disk (for filesystems).

While we intend to build this system to run natively on an embedded platform, we will test it on a UML virtual machine first. For this reason we need to create devices for the UML filesystems (though these will be redundant when the system runs on the actual embedded devices):

```
$ sudo mknod -m 660 0 b 98 udb/0
$ sudo mknod -m 660 1 b 98 udb/1
```

5.9 Administrative Files and Directories

Create a number of directories in the *$ROOT* directory:

```
$ cd $ROOT
$ mkdir boot root proc sys
```

The kernel and bootloader stages are stored in the */boot* directory. The */root* directory is the home directory for the root (superuser) account. The */proc* and */sys* directories are mounted points for the procfs and sysfs filesystems (see Sect. 3.2.1). The system's configuration reside, primarily, in the */etc* directory. Create subdirectories in the *$ROOT/etc* directory:

```
$ mkdir etc/{init.d,profile.d,network}
```

Create files *motd* and *issue*:

```
$ touch etc/{motd,issue}
```

The *motd* files contains a message of the day that is displayed when the user logs in and *issue* contains a message that is displayed prior to login. We leave it to the reader to populate these files.

Create subdirectories that will contain script files for configuring the network interface:

```
$ mkdir etc/network/{if-down.d,if-post-down.d,\
> if-pre-up.d,if-up.d}
```

Password authentication requires two files, namely, *passwd* and *group*. Most modern GNU/Linux systems include a shadow password file (*shadow*) for extra security. For convenience, we will not use a shadow password file.

The *passwd* file contains the valid user accounts for the system. There is one line of configuration per user. Each user must belong to a group. Groups are defined in the *group* file. We only need to define an account for the Superuser along with a corresponding group. Create a files *$ROOT/etc/passwd* and *$ROOT/etc/group* with the contents shown in Listings 5.2 and 5.3 respectively.

Listing 5.2 *Contents of /etc/passwd*

```
root::0:0:Superuser:/root:/bin/bash
```

Listing 5.3 *Contents of /etc/group*

```
root:!:0:
```

Glibc library is able to access database information from various sources using the name service switch (NSS). The *nsswitch.conf* file specifies the sources of these databases. Create the *$ROOT/etc/nsswitch.conf* with contents shown in Listing 5.4.

Listing 5.4 *Contents of /etc/nsswitch.conf*

```
passwd:                 files
```

```
group:                  files
shadow:                 files
hosts:                  files dns
networks:               files
protocols:              fIles
services:               files
```

Create the *$ROOT/etc/inittab* with contents shown in Listing B.1 in Appendix B.

Listing 5.5 shows the contents of *$ROOT/etc/fstab*. This is similar to the *fstab* in Listing 4.1 for the deb build except there is an additionel configuration line for the */var* directory. A temporary filesystem is mount on */var* in order to avoid exceessive write operations to the root filesystem in order to prolong the life of the solid state device.

Listing 5.5 *Contents of /etc/fstab*

```
proc /proc proc defaults 0 0
sysfs /sys sysfs defaults 0 0
tmpfs /dev/shm tmpfs defaults 0 0
tmpfs /tmp tmpfs defaults 0 0
tmpfs /var tmpfs defaults 0 0
```

The *services* file specifies the mapping between Internet services and transport (UDP and TCP) port numbers. The most convenient way to generate this file is copy it from the host:

```
$ cp /etc/services $ROOT/etc/
```

Listing 5.6 shows the content of *$ROOT/etc/network/interfaces* which specifies the configuration of the network interface (lo and eth0). Here we configure eth0 to get its IP details dynamically through DHCP. If you wish to configure static IP details use the example in Listing 4.5.

Listing 5.6 *Content of /etc/network/interfaces*

```
auto lo
iface lo inet loopback

auto eth0
iface eth0 inet dhcp
```

We need to create a script $ROOT/usr/share/udhcpc/default.script with the content shown in Listing 5.7. It will be necessary to create the *$ROOT/usr/share/udhcpc*

before creating the file:

```
$ mkdir $ROOT/usr/share/udhcpc
```

Listing 5.7 *Content of /usr/share/udhcpc/default.script*

```
#!/bin/sh

[ -z"$1" ] && echo "Error: call from udhcpc only" && exit 1

RESOLV_CONF="/etc/resolv.conf"
RESOLV_BAK="/etc/resolv.bak"

[ -n "$broadcast" ] && BROADCAST="broadcast $broadcast"
[ -n "$subnet" ] && NETMASK="netmask $subnet"

case "$1" in
    deconfig)
        if [ -f "$RESOLV_BAK" ]; then
            mv "$RESOLV_BAK" "$RESOLV_CONF"
        fi
        /sbin/ifconfig $interface 0.0.0.0
        ;;

    renew|bound)
        /sbin/ifconfig $interface $ip $BROADCAST $NETMASK

        if [ -n "$router" ]
        then
            for i in $router ; do
                route add default gw $i dev $interface
            done
        fi

        if [ ! -f "$RESOLV_BAK" ] && [ -f "$RESOLV_CONF" ]
        then
             mv "$RESOLV_CONF" "$RESOLV_BAK"
        fi

         echo -n > $RESOLV_CONF
         [ -n "$domain" ] && echo search $domain >>
         $RESOLV_CONF
         for i in $dns ; do
             echo nameserver $i >> $RESOLV_CONF
```

```
        done
        ;;
esac

exit 0
```

Create the file *$ROOT/etc/profile* with contents shown in Listing 5.8.

Listing 5.8 *Contents of /etc/profile*

```
#!/bin/bash

for i in /etc/profile.d/*
do
       . $i
done
```

In *$ROOT/etc/profile.d*, create the file: *env*, with the contents shown in Listing 5.9.

Listing 5.9 *Contents of /etc/env*

```
export PS1='\u@\h:\w\ $ '
```

During the course of its operation, GNU/Linux writes data to files in the */var* directory. The problem is that our root filesystem will be on a solid state device, namely, a compact flash card. Solid state devices have a much shorter *endurance* than conventional hard disks. If */var* is on the root filesystem, then continual writes to the directory will shorten the lifetime of the compact flash. It in not uncommon for the embedded devices to only mount the root filesystem read-only in order to protect the solid state device. Clearly this causes a problem when the system want to write to */var*. To overcome this problem, a tmpfs is mounted on *var*.

If you refer back to Listing 5.5, you will see the line below in */etc/fstab* which causes temporary filesystem to be mounted on */var*:

```
tmpfs /var tmpfs defaults 0 0
```

The system expects a directory structure below */var* but, as data is not persistent in a tempoary filesystem. For this reason we have to create the directory structure at boot-time. This is done in the bootmisc start-up script shown in Listing A.4 below. It does this by untarring a file once the temporary filesystem has been mounted on */var*.

Create a temporary directory in which to create the *var* directory structure:

```
$ mkdir $BASE/tmpd
$ cd $BASE/tmpd
```

Create the directory structure for *var*:

```
$ mkdir var/{,log/,run/,run/network/,tmp/}
```

Create a tar file of the directory structure and copy it to the $ROOT directory.

```
$ tar cf var.tar var/
$ cp var.tar $ROOT
```

5.10 Start-up Scripts

We need to write a number of start-up (and shutdown) scripts that are run by init. The start-up/shutdown scripts are located in the *$ROOT/etc/init.d* directory. These scripts are run according to the runlevel the system is in. So each runlevel there is an *rc* directory which contains symbolic links to the start-up/shutdown scripts relevant to that runlevel. Create the *init.d* and rc directories:

```
$ mkdir init.d rc{S.d,0.d,1.d,2.d,3.d,4.d,5.d,6.d}
```

For brevity, the code listings for all the start-up/shutdown scripts are in Appendix A. A list of the start-up/shutdown scripts are given below:

rcS The rcS script is the boot-time system initialization script which runs other start-up script. Listing A.1 shows the rcS script. The line below from the *$ROOT/etc/inittab* file (Listing B.1) causes this script to run:

```
si::sysinit:/etc/init.d/rcS
```

rc The rc scripts runs the */etc/inittab* file (Listing B.1) defines the system states under which the rc scripts are run. For example, the line below runs the rc script with the argument "2" for runlevel 2:

```
l2:2:wait:/etc/init.d/rc 2
```

Listing A.2 shows the rc script. The script runs a sequence of other scripts which can either start-up or shutdown a service.

mountall The mountall sctipt (in its start-up form) mounts all the filesystem listed in */etc/fstab*. For this system, these are the tmpfs filesystem and the procfs filesystem. It also re-mounts the root filesystem read-write (whereas, up to

this mount it was mounted read-only). In its shutdown form, mountall umounts all the filesystems. See Listing A.3.

bootmisc The bootmisc script. See Listing A.4.

hostname Configure the hostname of the system. See Listing A.5.

telnet Starts/stops the Telnet daemon. See Listing A.9.

ntp Starts/stops the network time protocol (NTP). See Listing A.10.

syslogd Starts/stops the syslog daemon. See Listing A.7.

klogd Start/stops the kernel log daemon. See Listing A.8.

halt Initiates a shutdown of the system. See Listing A.11.

reboot Initiates a reboot of the system. See Listing A.12.

sMake the start-up scripts executable:

```
$ cd $ROOT/etc
$ chmod +x init.d/*
```

We create symbolic links to the start-up/shutdown scripts in the rc directories. Links to start-up scripts begin with an “S” while links to shutdown scripts begin with a “K”. Create symbolic links in rcS.d:

```
$ cd $ROOT/etc/rcS.d
$ ln -s ../init.d/mountall S35mountall
$ ln -s ../init.d/bootmisc S37bootmisc
$ ln -s ../init.d/hostname S40hostname
$ ln -s ../init.d/network S40network
```

Create symbolic links in rc0.d:

```
$ cd $ROOT/etc/rc0.d
$ ln -s ../init.d/klogd K89klogd
$ ln -s ../init.d/syslogd K90syslogd
$ ln -s ../init.d/network K95network
$ ln -s ../init.d/mountall K97mountall
$ ln -s ../init.d/halt S99halt
```

Create symbolic links in rc2.d:

```
$ cd $ROOT/etc/rc2.d
$ ln -s ../init.d/syslogd S10syslogd
$ ln -s ../init.d/klogd S11klogd
```

Create symbolic links in rc6.d:

```
$ cd $ROOT/etc/rc6.d
```

```
$ ln -s ../init.d/klogd K89klogd
$ ln -s ../init.d/syslogd K90syslogd
$ ln -s ../init.d/network K95network
$ ln -s ../init.d/mountall K97mountall
$ ln -s ../init.d/reboot S99reboot
```

Just as we did with the deb system, we can carry out a preliminary test of this system using the chroot command:

```
$ sudo chroot $ROOT:
```

Set the root password (note, passwords are not echoed to the screen):

```
/ # passwd
Enter new UNIX password: letmein
Retype new UNIX password: letmein
passwd: password updated successfully
/ # exit
```

5.11 Test with UML

We aim to run this system natively on a physical CPU. We still have some work to do, but we should be able run the the system in its current form under UML. We can use the UML kernel we built in the previous chapter to test the emx root filesystem. We built the kernel with loadable module support, normally, we should install the kernel modules onto the root filesystem. The kernel, however, has sufficient drivers built in to get the system up and running for test purposes. We can, therefore, forego the module installation process.
Make a tape archive of the emx root directory:

```
$ cd $SRC
$ sudo tar zcvf root-emx.tar.gz root/
```

Create a 64 MByte for build a virtual filesystem on:

```
$ dd if=/dev/zero of=emx_fs seek=64 count=1 bs=1M
1+0 records in
1+0 records out
```

Format *emx_fs* as an ext2 filesystem (for brevity we omit the output of mkfs.ext2):

```
$ mkfs.ext2 -F emx_fs
```

Mount the filesystem as a loopback device:

```
$ sudo mount -o loop emx_fs root
```

Write the root directory structure to filesystem and unmount:

```
$ sudo tar zxvf root-emx.tar.gz
$ sudo umount root
```

Copy the *uml.sh* script into the current directory (which should be the $SRC directory). Create a link to the UML kernel built in Chap. 4:

```
$ ln ~/deb/linux-2.6.39.4/linux linux
```

With the emx system, we take a different approach to networking. Instead of using the host system as NATing router, we use it as a bridge and allow the UML VM to get its IP details dynamically from local DHCP server.

```
$ sudo dhclient ubridge
$ sudo brctl addif eth0
```

Run the emx system as a UML VM:

```
$ ./uml.sh -r emx_fs -n uth0
```

An extract from the console shows the allocation of IP address (along with other details) to Ethernet interface:

```
Bring up network interfaces: udhcpc (v1.19.3) started
Sending discover...
Sending select for 172.16.50.37...
Lease of 172.16.50.37 obtained, lease time 86400
done.
```

5.12 Running the System Natively

Having tested the system (so far) runs successfuly on a UML VM, we complete the the final steps so that we can run it natively on a i486 Soekris device.

5.12.1 Compiling the Kernel

We extracted the source file for the Linux kernel in Sect. 5.2 in order to build the kernel headers, therefore we can make the kernel source directory the working directory:

```
$ cd  ${BASE}/linux-2.6.35.13
```

Before we compile the kernel, we must generate a configuration file. The kernel comprises many options. Unfortunately, an explanation of all the options required for even a modest embedded system is outside the scope of this book. See [2] for a general description of Linux kernel options. Download the kernel configuration for a Soekris net4521:

```
$ wget www.thinglabs.co.uk/emx/config-2.6.35.13
```

Copy the configurqtion file to *.config*:

```
$ cp  config-2.6.35.13 .config
```

Configure the kernel:

```
$ make menuconfig
```

While do not intend to go into kernel options is detail, it is worth noting that the loadable module support option is disabled. This means that all device driver will be built statically into the kernel itself. This simplifies the kernel build as it will the obviate steps required to compile and install the modules. Compile the kernel:

```
$ make
```

Install the kernel onto the root filesystem

```
$ cp System.map $ROOT/boot/System.map-2.6.35.13
$ cp .config $ROOT/boot/config-2.6.35.13
$ cp arch/i386/boot/bzImage $ROOT/boot/bzImage-2.6.35.13
```

5.12.2 The Filesystem

The Soekris uses a compact flash as its hard drive. We summerise the steps below for preparing the compact flash below:

- Create the partition table (fdisk).
- Format the partition as an ext2 filesystem.

- Install the root directory structure.
- Install the bookloader.

First we must connect the compact flash to the host system. For this we need a compact flash reader. These usually connected to the host system via USB. Before connecting the compact flash reader (with the compact flash inserted) to the host check the */proc/partitions* file. On our system the content of the file looks like this (it is likely the output will be different on your system):

```
$ cat /proc/partitions
major minor  #blocks  name

   8        0  156290904 sda
   8        1     102400 sda1
   8        2   39062500 sda2
   8        3          1 sda3
   8        5  113282316 sda5
   8        6    3839503 sda6
```

Connect the compact flash reader to the host computer and repeat the command above. You should see some additional lines (how many, will depend upon number of partitions already configured on the compact flash card):

```
   8       16      62720 sdb
   8       17      62713 sdb1
```

The output above tells us that the compact flash device is */dev/sdb* and it has a single partition (*/dev/sdb1*). The operating system may be configured to automatically mount filesystem when the drive is connected to the host. If this is the case, then unmount the filesystems:

```
$ sudo umount /dev/sdb1
```

First we create a partition table using fdisk:

```
$ sudo fdisk -uc /dev/sdb
```

Check for any existing partition table entries:

```
Command (m for help): p

Disk /dev/sdb: 64 MB, 64225280 bytes
2 heads, 62 sectors/track, 1011 cylinders, total 125440
sectors
```

```
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x00000000

   Device Boot    Start       End    Blocks  Id  System
```

Create a partition:

```
Command (m for help): n
Command action
   e   extended
   p   primary partition (1-4)
```

Select a primary partition

```
p
```

Select 1 as the partition number and hit return at each subsequent prompt:

```
Partition number (1-4): 1
First sector (2048-125439, default 2048):
Using default value 2048
Last sector, +sectors or +size{K,M,G} (2048-125439,
default 125439):
Using default value 125439
```

Make the partition bootable and select 1 when prompted for the partition:

```
Command (m for help): a
Partition number (1-4): 1
```

Check the details of the partition created:

```
Command (m for help): p

Disk /dev/sdb: 64 MB, 64225280 bytes
2 heads, 62 sectors/track, 1011 cylinders, total
125440 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x00000000
```

```
   Device Boot    Start       End    Blocks  Id  System
/dev/sdb1   *      2048    125439     61696  83  Linux
```

Write the partition table:

```
Command (m for help): w
The partition table has been altered!

Calling ioctl() to re-read partition table.
Syncing disks.
```

Format the partition:

```
$ sudo mkfs.ext2 /dev/sdb1
```

Mount the compact flash drive on the root directory:

```
$ sudo mount /dev/sdb1 root
```

Untar the root directory structure onto the filesystem:

```
$ sudo tar zxvf root-emx.tar.gz
```

We need to make a small modification to the *inittab* file. The Soekris does not have physical hardware for the tty[1–6] devices. Lines in the *$ROOT/etc/inittab* file like the one below, will cause init to respawn getty processes too quickly:

```
2:23:respawn:/sbin/getty 38400 tty2
```

As a result, the message below will be displayed on the console periodically:

```
INIT: Id "2" respawning too fast: disabled for 5 minutes
```

In order to prevent this rapid respawning (and the subsequent messages), comment out the lines in the *$ROOT/etc/inittab* file that respawn gettys on tty[1–6], see below:

```
#1:2345:respawn:/sbin/getty 38400 tty1
#2:23:respawn:/sbin/getty 38400 tty2
#3:23:respawn:/sbin/getty 38400 tty3
#4:23:respawn:/sbin/getty 38400 tty4
#5:23:respawn:/sbin/getty 38400 tty5
#6:23:respawn:/sbin/getty 38400 tty6
```

Obviously, you can omit the steps above if you are using a platform with physical hardware for these tty devices.

5.12.3 The Bootloader

Syslinux is collection of lightweight bootloaders for various filesstem types. The Syslinux bootloader itself is for MS-DOS filesystems only. While the root filesystem is ext2, we could create an MS-DOS *boot* filesystem which holds the Syslinux bootloader files. However, in order to reduce the complexity of having two filesystems, we use the Extlinux version of the bootloader that works with ext filesystems. Download the Syslinux package to the *$BASE* directory:

```
$ cd $SRC
$ wget http://www.kernel.org/pub/linux/utils/boot/\
> syslinux/syslinux-4.05.tar.gz
```

Syslinux needs the header files from the UUID package:

```
$ sudo apt-get install uuid-dev
```

Extract the source from the tar file and make *syslinux-4.05* the working directory:

```
$ tar zxvf syslinux-4.05.tar.gz
$ cd syslinux-4.05
```

Compile Syslinux (this compiles all the bootloaders):

```
$ make
```

Make *extlinux* the working directory:

```
$ cd extlinux
```

Install the bootloader (ensure the ROOT variable is set):

```
$ ./extlinux --install $ROOT/boot/extlinux
```

The bootloader needs a configuation file: *$ROOT/boot/extlinux/extlinux.conf*. Create this file with the content shown in Listing 5.10

Listing 5.10 *Contents of /boot/extlinux/extlinux.conf*

```
SERIAL 0 19200
```

Fig. 5.2 Soekris motherboard showing compact flash card

```
DEFAULT linux
LABEL linux
SAY Now booting the kernel from SYSLINUX...
KERNEL /boot/bzImage-2.6.35.13
    APPEND ro root=/dev/hda1 console=ttyS0,19200n8
TIMEOUT 30
PROMPT 1
```

The system is now complete and we unmount the CF drive:

```
$ cd $BASE
$ sudo umount root
```

5.12.4 Booting the System

We are now ready to boot the system from the compact flash card. The compact flash socket is located inside the Soekris so it is necessary to remove the cover (Fig. 5.2). Plug the compact flash card into the compact flash socket (then replace the cover). We then need to connect up the Soekris to a host and the local Ethernet network as shown in Fig. 5.3. In our caae, the host machine was a laptop (running GNU/Linux).

The serial port of the Soekris is connected to the serial port of the host. The init program is configured to run a getty on the Soekris unit's serial 0 (ttyS0). If we run a terminal emulator on the host we can gain access to Soekris though a serial link. As both the host and the Soekris are data terminal equipment (DTE) we need the serial cable needs to be a *null modem*. Table 5.1 shows the pin-out configuration for both

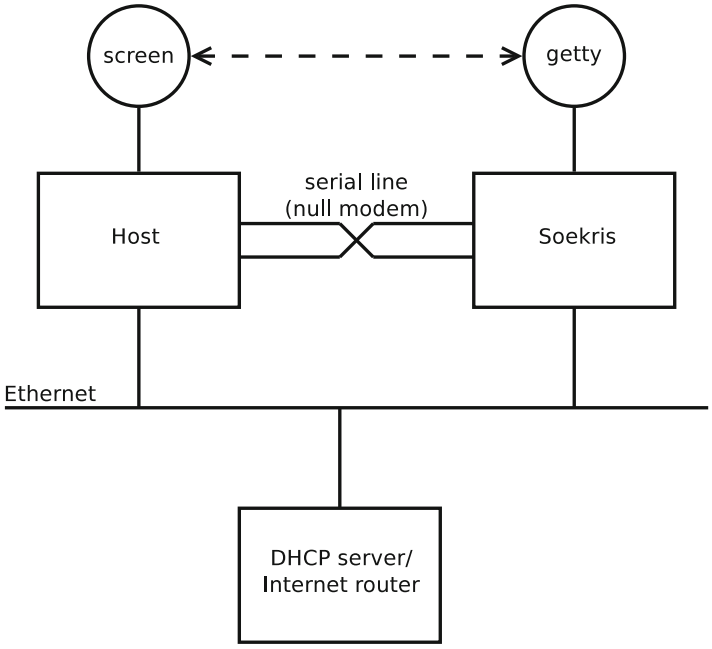


Fig. 5.3 Soekris configuration

Table 5.1 Null modem

Signal	From		To		Description
	25-pin	9-pin	25-pin	9-pin	
Grd	1	–	1	–	Ground
Tx	2	3	3	2	Transmit
Rx	3	2	2	3	Receive
RTS	4	7	5	8	Request to send
CTS	5	8	4	7	Clear to send
Sig Grd	7	5	7	5	Signal ground
DSR	6	6	20	4	Data set ready
DCD	8	1	20	4	Data carrier ready
DTR	20	4	6/8	6/1	Data terninal ready

25- and 9-pin connectors. However, rather than making a null modem cable it would be just as easy to purchase one from an electrical retailer. As with mant laptops these days, our did not have serial port. We, therefore, had to use one of the USB ports with a USB-serial converter cable.

Power up the Soekris and run the command below:

```
$ screen /dev/ttyUSB0 19200,cs8
```

Statements from the boot procedure will be displayed on the screen. Once the kernel has booted, it runs the init process, which in turn starts the systems services:

```
udhcpc (v1.19.3) started
Sending discover...
Sending select for 172.16.50.38...
Lease of 172.16.50.38 obtained, lease time 86400
done.
INIT: Entering runlevel: 2
Starting syslogd: syslogd started sucessfully.
Starting klogd: klogd started sucessfully.
Starting telnetd: telnetd started sucessfully.
```

Once the boot up procedure has completed the login prompt is displayed. Login as root in the usual way. We can also login over the network using Telnet:

```
$ telnet 172.16.50.38
```

5.13 Summary

We have looked at two methods of building an embedded GNU/Linux operating systems. The deb system described in Chap. 4 was generated using the debootstrap utility. We ran the system as a UML VM but it would have been fairly straight forward get it working on an actual hardware platform. The debootstrap utility is not specifically for generating embedded systems, it can be used for general computing systems such as desktops, laptops and servers.

We built the emx system manually, compiling all the sources code and creating all the necessary subdirectories, system files and devices. The emx system was tested with UML but we took the system a stage further and ran it on a Soekris embedded platform.

While the deb system is quite small it is, nevertheless, somewhat bloated for an embedded system. It does, however, offer the flexibility of adding packages post-installation through the Debian package management utility (Apt).

The emx system, in contrast, is much smaller. Consequently, it has fewer features and is quite limited in its usefulness. Unlike the deb system, however, utilities can be added to the system but this cannot be done dynamically while the system is running due to the lack of package manager. New software has to compiled from source then installed on the root filesystem. This can be done direct to the compact flash card (or whatever device is being used as a hard drive) provided it is mounted on the host system. Compiling software from source can be fraught with problems.

Most packages are dependent upon software libraries from other packages (which in turn can be dependent upon other libraries). Compiling a particular package can, therefore, involve having to build many dependencies. Package managers like Apt resolve dependency issues automatically. Fortunately, many GNU/Linux utilities are supported by Busybox albeit they are usually lightweight versions of the originals.

The emx system's compact size is not necessarily an advantage over deb. We were able to fit the emx root filesystem on a 64 MByte compact flash. However, these days, it is difficult to buy solid state drives this small. At the time of writing this book a browse through the Amazon web site revealed the smallest compact flash card was 4 GByte, which is more than enough to accommodate the deb root filesystem.

References

1. Soekris Engineering (2013) Soekris engineering. http://soekris.com/products.html. Accessed 14 May 2013
2. Kroah-Hartman G (2009) Linux Kernel in a Nutshell. O'Reilly Media, Sebastopol

Compiler Toolchains

6

Most software is written in high level programming languages such as C, Java, Perl or Python. High-level languages comprise machine independent instructions which cannot be directly executed by the processor. High-level language instructions, therefore, must be translated into machine level (or machine dependent) instructions.

High level languages are often categorised as either compiled or interpreted. A compiled language is translated from source code to object code prior to execution using a program called a *compiler*. The compiler produces an executable file from a file (or files) containing the source code.

With interpreted languages, the source code is translated and executed line by line. Interpreted languages are more convenient than compiled languages. They are more portable, because code can be executed on any machine architecture that has the interpreter. However, the execution of interpreted code tends to be slower, because (high-level) instructions must be translated (into low level instructions) each time they are executed.

This distinction between compiled and interpreted languages has blurred somewhat with the arrival of languages such as Java and Python. These languages are compiled into a *bytecode* which is then executed by an interpreter.

The compiler translates high-level instructions into machine code in a number of stages. Compilers, therefore, are rarely a single program, but rather a collection of programs. This collection of programs is called a *toolchain* because they are each run as a chain of commands, each passing its output to the next program in the chain. The final program in the chain will generate the object code in a file. The main stages (programs) in a compiler toolchain are:

- Preprocessor
- Code generator
- Assembler
- Linker

As this book is about embedded operating systems we are interested in a particular type of toolchain, namely a *cross toolchain*. Given that embedded systems are limited

A. Holt and C.-Y. Huang, *Embedded Operating Systems*, Undergraduate Topics in Computer Science, DOI 10.1007/978-1-4471-6603-0_6

in resources (typically) it is impractical to compile source code for an embedded system on the system itself. For this reason, source code compilation is performed on a host machine which may have different processor architecture to the embedded system. A cross compiler toolchain is a toochain that runs on a host system but builds software for a target system of a different architecture. In this chapter we discuss compiler toolchains. We restrict our discussion to the GNU compiler collection (GCC) as to is the de facto toolchain for the GNU/Linux system (though it is not exclusive to GNU/Linux). In the last part of the chapter we show how to build a cross compiler toolchain.

6.1 GCC

GCC (GNU compiler collection) is a set of compilers. GCC was originally just a C compiler (and was called the GNU C compiler), however it now can compile other languages, such as C++, Fortran, Pascal and Java.

GCC is the primary compiler for GNU/Linux (as well as BSD derivatives and Mac OS X). The use of GCC is not confined to open source projects, it is also used for commercial and proprietary development.

In the following example we show how to compile the ubiquitous "hello world" program. The C source code is shown in Listing 6.1.

Listing 6.1 *hello.c*

```
#include <stdio.h>

int main ()
{
     printf("hello world\n");
     return 0;
}
```

Compile the source *hello.c* with GCC:

```
$ gcc -Wall -o hello hello.c
```

This produces a binary file *hello* which contains the binary executable code. We can run the resultant program, thus:

```
$ ./hello
hello world
%
```

In the example above, GCC has performed all the "compilation" stages, that is, the preprocessing, compilation, assembly and linking. It is possible to terminate the procedure at intermediate stages using GCCs options.

6.1.1 The Preprocessor

C, like a number of languages, requires a preprocessing phase. The purpose of the preprocessor is to remove the features of the source code that improves its readability. The GCC pre-processor is (`cpp`). The preprocessor performs the following tasks:

- Removes comments (replacing them with a single space).
- Expands any macro definitions.
- Removes any line breaks unless they are in literals.
- Replaces `#include` lines with the respective file contents.
- Re-assemble lines the programmer has split up for clarity.
- Tokenizes keywords and builds table of symbols which might be used for code generation.

We demonstrate the role of the preprocessor in the next example. Listing 6.2 shows a simple C program *incr.c*.

Listing 6.2 *Content of incr.c*

```
/* incr.c
   Preprocessor example
*/

#include"incr.h"

main() {

    int x = 0; /* declare an integer x */

    /* increment the value 2 and assigns to x */
    x = \
        incr(2);

}
```

The `#include` directive tells the preprocessor to load the content of the specified file at the point the directive appeared in the original source file. The text delimited by `/*` and `*/` are comments. In this case, the include file is *incr.h* shown in Listing 6.3.

Listing 6.3 *Content of incr.h*

```
/* incr.h
   Defines increment macro
*/

#define incr(n) n + 1
```

The `#define` directive specifies a macro, in this case called `incr`. The macro takes and argument (just like a function call) to which 1 is added. Also, the assignment of `x` is split over two lines. This is unnecessary but we want to demonstrate that how the preprocessor reassembles lines.

Two file names are passed are arguments to cpp. The first file is the infile and contains the initial C source code to be preprocessed (in this case *incr.c*. Any files specified in `#include` directives of the infile will also be processed. The second file is outfile which cpp writes the resultant preprocessed code. Run cpp, thus:

```
$ cpp incr.c incr.cpp
```

The command-line above produces an output file *incr.cpp*, the content of which is shown in Listing 6.4.

Listing 6.4 *Content of incr.cpp*

```
# 1 "incr.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "incr.c"
# 1 "incr.h" 1
# 2 "incr.c" 2



main() {

    int x = 0;


    x = 2 + 1;


}
```

We can see that the `cpp` has performed a number of preprocessing tasks. Firstly, the code from *incr.h* has been included. The comments have been removed and replaced them with a single space. The assignment statement of `x`, which was split over two lines, has been re-assembled into one. Finally, the macro statement `incr(2)` has been replaced with the statement: `x = 2 + 1`.

6.1.2 Optimisation

The purpose of the optimisation step is to try and maximises or minimise some attribute of the the executable code, for example:

- Size of the binary image.
- Execution speed.
- Memory usage.
- Energy consumption (this is of particular relevance embedded systems).

In addition, the optimisation process may reduce the actual execution time and memory use of the compilation itself. Optimisation is optional step in the compilation of the code. GCC support a number of levels of optimisation which can be through command-line options. See Table 6.1 for a description of GCC optimisation levels. Here we present a simple example of code optimisation. We compare optimisation level 1 with level 0 (no optimisation) using code in Listing 6.5.

Table 6.1 GCC optimisation options

Level	Option	Comment
0	`-O0`	No optimisation. This is the same as omitting the `-O` option
1	`-O1` or `-O`	Performs optimisation functions that do not require any speed/space trade-off. The size of the binary image should be smaller and its execution speed fastercompared to the `-O0` level. Furthermore the compilation can be shorter
2	`-O2`	Performs additional optimisation (over that of `-O1`) that do not require any speed/space trade-off. Executions speeds should be faster without increasing the size of the image. Compilation times could be slower
2.5	`-Os`	Performs optimisation aimed at reducing the size of the generated code. In some cases this can also improve performance because there is less code to execute
3	`-O3`	The aim of this level of optimisation is to increase execution speed. It does this at the cost of increasing the binary image size. In some cases, however, the increase in code can have the effect of reducing performance
3	`-Ofast`	Performs optimisations that disregards standards compliance

Listing 6.5 *Optimisation example: content of opt.c*

```
#include <stdio.h>

#define N 100000

int main()
{
    long i, j, k = 0;

    for(i = 0; i < N; i++)
        for(j = 0; j < N; j++)
            k += (i - j);


    printf("k = %ld\n", k); /* k = 0 */
}
```

Compile the *opt.c* for optimisation levels 0 and 1:

```
$ gcc -O0 -o opt0 opt.c
$ gcc -O1 -o opt1 opt.c
```

Run the executable for optimisation level 0:

```
$ time ./opt0
k = 0

real    0m31.729s
user    0m31.702s
sys 0m0.000s
```

Now run the executable for level 1:

```
$ time ./opt1
k = 0

real    0m9.080s
user    0m9.065s
sys 0m0.004s
```

We can see that optimised code executes in about a third of the time compared to the none optimised code. We leave to the reader to test the code execution time for optimisation levels 2 and 3.

6.1.3 The Assembler

Assembly language is a low-level programming language where machine language operations are represented by mnemonics. There is (usually) a one-to-one correspondence between numerical machine language operations and assembly language mnemonic instructions. The GCC code generator converts high-level language instructions to assembly code. This code is passed to the assembler which translates the assembly language instructions in machine code.

The C source code in Listing 6.6 declares a variable `x` and assigns the value 42 to it. The function `main` returns the value of `x`.

Listing 6.6 *Content of file decl.c*

```
int main() {
    int x = 42;
    return x;
}
```

Compile *decl.c* but stop the process at the assembler stage (`-S`):

```
$ gcc -Wall -S decl.c
```

This produces assembly code in the file *decl.s* shown in Listing 6.7.

Listing 6.7 *Content of file decl.s*

```
.file   "decl.c"
.text
.globl main
.type   main, @function
main:
pushl   %ebp
movl    %esp, %ebp
subl    $16, %esp
movl    $42, -4(%ebp)
movl    -4(%ebp), %eax
leave
ret
```

```
.size   main, .-main
.ident  "GCC: (Ubuntu 4.4.3-4ubuntu5.1) 4.4.3"
.section    .note.GNU-stack,"",@progbits
```

Run the assembler to translate the assembly language instructions into object code (*decl.o*):

```
$ as -o decl.o decl.s
```

We still need to perform the linker stage on the object code, so we call gcc to complete the compilation process:

```
$ gcc -o decl decl.o
```

The binary executable and check the return code:

```
$ ./decl
$ echo $?
42
```

6.1.4 The Linker

Large Programs are normally broken into multiple source files for convenience. These source files are combined and compiled together to form a single executable. For very large software development projects, it is convenient to compile into separate object code blocks. While object code comprises machine dependent instructions, it cannot be executed. The toolchain has to perform the final stage and *link* the separate object code blocks into a single executable. The advantage of organising the software into separate object code blocks is that if a change is made to one part of the (source) code, only the object code block that is affected needs to be compiled rather than the entire code. Object code files have the suffices `.o`. The linker can also link object code from libraries. There are two varieties of library:

- Static libraries (suufix `.a`).
- Dynamic shared libraries (suffix `.so`).

In the previous section we ran the assembler to translate assembly instructions into machine dependent code. We had to run the linker (gcc with the `-o` switch) to generate the final executable binary code. This is because each object code module's address starts at zero. The linker combines these various object code modules and performs code *relocation*. If we compare *decl.o* and *decl*, we see that *decl.o* contain relocatable code whereas *decl* contain executable code:

```
$ file decl.o decl
decl.o: ELF 32-bit LSB relocatable, Intel 80386, version
1 (SYSV), not stripped
decl:   ELF 32-bit LSB executable, Intel 80386, version
1 (SYSV), dynamically linked (uses shared libs), for
GNU/Linux 2.6.15, not stripped
```

The function of the linker is demonstrated in the example below:

```
$ gcc -c -o mean.o mean.c
```

The source code for *mean.c* is shown in Listing 6.8. The command above generates a file *mean.o* which contains the compiled (but not linked) object code.

Listing 6.8 *Content of file mean.c*

```
double mean(double *x, int n)
{

    int i;
    double sum;

    for(i=0;i<n;i++)
         sum += x[i];

    return sum/n;
}
```

The source code in Listing 6.9 shows the program, analysis, which calls the arithmetic mean function in *mean.c*. Compile *analysis.c* into relocatable object code:

```
$ gcc -c -o analysis.o analysis.c
```

Listing 6.9 *Content of file analysis.c*

```
#include <stdio.h>
#include <stdlib.h>

#define N 100

extern double mean(double *, int);
```

```
int main (int argc, char **argv)

{
    int i,n;
    double m = 0.0;
    double x[N];

    if(argc<2) {
        printf ("usage: %s <list values>\n", argv[0]);
        exit(-1);
    }

    n = argc - 1;

    for(i=0;i<n;i++)
        x[i] = atof(argv[i+1]);

    m = mean(x,n);
    printf("mean: %f\n", m);

    return 0;
}
```

We link *analysis.o* and *mean.o* to create the *analysis* binary executable file. As *analysis.o* and *mean.o* are object files, GCC only performs the final linking stage to complete the compilation:

```
$ gcc -Wall -o analysis analysis.o mean.o
```

Verify the resultant executable:

```
$ ./analysis 1 2 3 4 5 6
mean: 3.500000
```

In the example above we linked two object files, namely *analysis.o* and *mean.o*, to form the binary executable file, *analysis*. If we had many such object files (and typically we might) then linking them all could get somewhat unwieldy. Object files can be compiled into a single archive called a *library*. Libraries come in two forms, namely, static and shared.

Static libraries are created using the archiver utility, ar (which is part of the toolchain). Static library file have the prefix "lib" and the suffix ".a". As with regular object files, static libraries are linked to the program code at compile time.

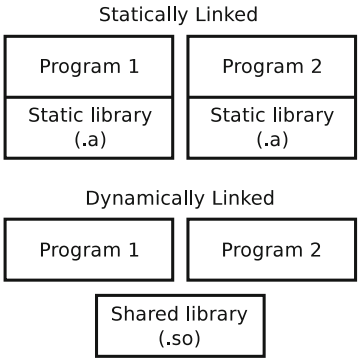


Fig. 6.1 Static versus dynamic libraries

Shared libraries are libraries that are loaded and linked dynamically by programs when they start (rather than at compile time). When a binary executable runs, the program loader runs. The loader locates and loads all the shared libraries referenced by the executable. The loader locates shared libraries by searching a number of directories. These directories are specified in */etc/ld.so.conf*. Recall we had to create the file */etc/ld.so.conf* in Sect. 5.2. Shared library directories can also be specified in the `LD_LIBRARY_PATH` environment variable. Shared library files have a "lib" prefix and a ".so" suffix.

The advantage of shared libraries is that they save memory space. Code from static libraries are stored in the files of the final executable whereas as the code from shared libraries are not. Furthermore, with shared libraries, only one copy of the library code has to be resident in primary memory regardless of how many running programs use it. That is, the library code is *shared*. For each running program that uses static library, there is an instance of it in primary memory. The diagram in Fig. 6.1 illustrates the difference between static and shared libraries.

We present examples of both static and shared libraries. Using the ar command. We create a static library of *statistics* functions. Create a static library using the *mean.o* object file:

```
$ ar r libstat.a mean.o
ar: creating libstat.a
```

Note, the file for the stat[1] library needs to be prefixed with "lib" and suffixed with ".a". The command below shows the modules in the stat library, confirming it contains *mean.o* (only):

```
$ ar t libstat.a
mean.o
```

[1] Note that here we use "stat" to denote *statistics* rather than *static* albeit stat is a static library.

Now we show how to add a variance function to the stat library. First we create a source code file *var.c* as shown in Listing 6.10.

Listing 6.10 *Content of file var.c*

```
#include <math.h>

double var(double *x, int n)
{

    int i;
    double sum = 0.0, sqdev = 0.0, mu;

    for(i=0;i<n;i++)
         sum += x[i];

    mu = sum/n;

    for(i=0;i<n;i++)
         sqdev += pow(x[i] - mu, 2.0);

    return sqdev/(n-1);
}
```

Compile the source into object code:

```
$ gcc -Wall -c -o var.o var.c
```

Add the *var.o* module to the stat library:

```
$ ar r libstat.a var.o
$ ar t libstat.a  # display modules in stat libray
mean.o
var.o
```

Now we can extend the functionality of the analysis program to calculate variance as well as the mean. The file *analysis.c* is modified to use the variance function defined in *var.c*. The modified code is shown in Listing 6.11.

Listing 6.11 *Modified content of file analysis.c*

```
#include <stdio.h>
#include <stdlib.h>
```

```
#define N 100

extern double mean(double *, int);
extern double var(double *, int);

int main (int argc, char **argv)

{
    int i,n;
    double m = 0.0, v = 0.0;
    double x[N];

    if(argc<2) {
        printf ("usage: %s <list values>\n", argv[0]);
        exit(-1);
    }

    n = argc - 1;

    for(i=0;i<n;i++)
        x[i] = atof(argv[i+1]);

    m = mean(x,n);
    v = var(x,n);
    printf("mean: %f var: %f\n", m,v);

    return 0;
}
```

Re-compile *analysis.c* with the stat library:

```
$ gcc -Wall -L. -o analysis analysis.c -lstat
```

The stat library is linked to the analysis program using the `-l` option. We also need to tell GCC the location of library with the `-L` option. Verify the results of the analysis program:

```
$ ./analysis 1 2 3 4 5 6
mean: 3.500000 var: 3.500000
```

In this example, we compile the analysis program using a shared library. Build the shared library with the command-line below:

```
$ gcc -shared -o libstats.so mean.o var.o
```

We call this library "stats" in order the distiguish it from the static library "stat" that we built earlier. Compile analysis using the stats shared library:

```
$ gcc -o analysis -L. analysis.c -lstats
```

If we run the ldd command on the *analysis* executable file we can see that the stats shared library is referenced:

```
$ ldd analysis
libstats.so => ./libstats.so (0xb785f000)
libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb76e7000)
/lib/ld-linux.so.2 (0xb7864000)
```

If we run the analysis program, it throws an error:

```
$ ./analysis 1 2 3 4 5 6
./analysis: error while loading shared libraries:
libstats.so: cannot open shared object file: No such
file or directory
```

We need to inform the loader of the location of the stats shared library. We could do this by editing the system's */etc/ld.so.conf* file (or any include files referenced within it). For convenience, we use the `LD_LIBRARY_PATH` environment variable. We add the current directory to `LD_LIBRARY_PATH` and export it:

```
$ export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
```

Now the loader can locate the stats shared library:

```
$ ./analysis 1 2 3 4 5 6
mean: 3.500000 var: 3.500000
```

6.2 Build an ARM Cross Toolchain

Embedded systems are typically low in processing speed and memory (primary and secondary). It is, therefore, unlikely that compilation software will be included as part of an embedded system's own operating system. Embedded system development is performed (typically) on more powerful systems designed for that purpose. Here,

it is important to distinguish between the *host* and the *target* platform. The host platform is the system on which the toolchain runs. The binary images of toolchain itself must be built for the processor architecture of the host. The toolchain builds binary images for a *target* platform. In the examples shown above (in this chapter and previous chapters), the host and target platforms are the same (in this case, an Intel 386 architecture). Where host and target are different, A *cross* toolchain is required.

In this section we show how to build an ARM cross toolchain. The toolchain runs on an i386 host and build binaries for the ARM target. Cross toolchains are notoriously difficult to build. While the toolchain described in this section will successfully compile C source to ARM binary code, it is merely an academic exercise. We do not recommend you use it as a production toolchain. The basic procedure for building this cross compiler toolchain is as follows:

- Build the Binutils package.
- Build a GCC bootstrap compiler.
- Build C library (we use Newlib in this example rather than Glibc).
- Buidl the final (cross) GCC toolchain.

A number of packages for the host system are required and should be downloaded using the appropriate package manager. For Debian based distributions, use:

```
$ sudo apt-get install -y flex bison libgmp3-dev \
> libmpfr-dev autoconf texinfo
```

We approach the development of the cross compiler toolchain in the same way as our other examples in that create a number of subdirectories to the build the software and use environment variables to handle options. Create a directory structure to work in:

```
$ mkdir armcc/{,toolchain}
```

Set the environment variables, BASE and TOOLCHAIN, to point to these directories:

```
$ export BASE=$HOME/armcc
$ export TOOLCHAIN=$BASE/toolchain
```

Set the TARGET to “arm-elf”

```
$ export TARGET=arm-elf
```

Table 6.2 Binutils

Command	Description
as	Assembler
ld	Linker
gprof	Profiler
addr2line	Convert addresses into file names and line numbers
ar	Create, modify, and extract from archives
c++filt	Demangling filter for C++ symbols
dlltool	Creation of Windows dynamic-link libraries
gold	Alternative linker
nlmconv	Object file conversion to a NetWare Loadable Module
nm	List symbols from object files
objcopy	Copy object files, possibly making changes
objdump	Display information about object files
ranlib	Generate index to archive
readelf	Display content of ELF files
size	List section and total sizes
strings	Print strings of characters in files
strip	Remove symbols from object files

6.3 Binutils

The Binutils package is a collection of tools for creating and manipulating binary images files, libraries and assembler files. The tools in Binutils are described in Table 6.2.

Change the working directory:

```
$ cd $BASE
```

Download the Binutils source code:

```
$ wget http://ftp.gnu.org/gnu/binutils/\
> binutils-2.19.1.tar.gz
```

Extract the source code from the tar file:

```
$ tar zxvf binutils-2.19.1.tar.gz
```

Create a directory in which to build Binutils:

```
$ mkdir build_binutils
```

Make *build_binutils* the current directory:

```
$ cd build_binutils
```

Set the configure options in a environment variable array:

```
$ unset OPTS
$ OPTS[0]="--target=$TARGET"
$ OPTS[1]="--prefix=$TOOLCHAIN"
$ OPTS[2]="--enable-interwork"
$ OPTS[3]="--enable-multilib"
$ OPTS[4]="--with-gnu-as"
$ OPTS[5]="--with-gnu-ld"
$ OPTS[6]="--disable-nls"
```

Configure Binutils:

```
$ ../binutils-2.19.1/configure ${OPTS[*]}
```

Compile and install:

```
$ make && make install
```

This completes the Binutils build.

6.3.1 GCC (Bootstrap Compiler)

We need a C library in order to build the GCC cross compiler. For the puprose of this exercise, we use Newlib. However, we need a cross compiler in order build Newlib. We resolve this apparent paradox by building a *bootstrap* compiler. The bootstrap compiler does not require a C library and providing us a basic C compiler that we can use to build the C library. Download the source code for GCC:

```
$ wget http://ftp.gnu.org/gnu/gcc/gcc-4.3.2/\
> gcc-4.3.2.tar.gz
```

Extract the source code from the tar file:

```
$ tar zxvf gcc-4.3.2.tar.gz
```

Create a build directory for GCC and make it the current directory:

```
$ mkdir build_gcc1 && cd build_gcc1
```

Set the configure options in an environment variable array:

```
$ unset OPTS
$ OPTS[0]="--srcdir=../gcc-4.3.2"
$ OPTS[1]="--prefix=$TOOLCHAIN"
$ OPTS[2]="--target=arm-elf"
$ OPTS[3]="--with-newlib"
$ OPTS[4]="--disable-threads"
$ OPTS[5]="--enable-multilib"
$ OPTS[6]="--enable-interwork"
$ OPTS[7]="--disable-shared"
$ OPTS[8]="--with-gnu-as"
$ OPTS[9]="--with-gnu-ld"
$ OPTS[10]="--disable-nls"
$ OPTS[11]="--enable-languages=c"
$ OPTS[12]="--without-headers"
```

Configure GCC and compile:

```
$ cd build_gcc1
$ ../gcc-4.3.2/configure ${OPTS[*]}
$ make all-gcc
```

Install the bootstrap compiler:

```
$ make install-gcc
```

6.3.2 Newlib

Now that we have a bootstrap compiler, we can build our C library. Download the Newlib source:

```
$ cd $BASE
$ wget ftp://sources.redhat.com/pub/newlib/\
> newlib-1.9.0.tar.gz
```

Extract the source from the tar file:

```
$ tar zxvf newlib-1.19.0.tar.gz
```

Set the options:

```
$ unset OPTS
$ OPTS[0]="--prefix=$TOOLCHAIN"
$ OPTS[1]="--target=arm-linux"
$ OPTS[2]="--enable-interwork"
$ OPTS[3]="--enable-multilib"
$ OPTS[4]="--with-gnu-as"
$ OPTS[5]="--with-gnu-ld"
$ OPTS[6]="--disable-nls"
```

Create a directory n which to build Newlib and make it the current directory:

```
$ mkdir build_newlib && cd build_newlib:
```

Configure:

```
$ ../newlib-1.19.0/configure ${OPTS[*]}
```

Compile and install Newlib:

```
$ make && make install
```

6.3.3 GCC

In this section we build the full cross compiler to complete the toolchain. We will use the same source code we used to build the bootstrap compiler, so it is a good idea to return the code to its original *clean* state:

```
$ cd $BASE/gcc-4.3.2/
$ make mrproper && cd ../build_gcc2
$ cd ..  # return to the $BASE directory
```

Set the options:

```
$ unset OPTS
$ OPTS[0]=--target=$TARGET
$ OPTS[1]=--prefix=$TOOLCHAIN
$ OPTS[2]=--enable-interwork
$ OPTS[3]=--enable-multilib
$ OPTS[4]=--with-gnu-as
$ OPTS[5]=--with-gnu-ld
$ OPTS[6]=--with-newlib
```

```
$ OPTS[7]=--disable-shared
$ OPTS[8]=--enable-languages="c,c++"
```

Create *build_gcc2* and make it the current directory:

```
$ mkdir build_gcc2 && cd build_gcc2
```

Configure:

```
$ ../gcc-4.3.2/configure ${OPTS[*]}
```

Finally, compile and install GCC:

```
$ make && make install
```

6.4 Testing the Toolchain

Having successfully built the ARM cross toolchain, we demonstrate that it works. The use the toolchain to compile the "hello world" program, shown in Listing 6.1, to an ARM binary. However, once the source code has been compiled we need an ARM platform to run it on.

We use the ARM instruction simulator (ARMulator) in the GNU debugger (GDB). The ARMulator is able to simulate the behaviour of a number of ARM cores. While debuggers are not strictly part of a compiler toolchain, they are a useful tool for testing programs. A detailed description of how to use GDB as debugging tool is beyond the scope of this book. We use it here, merely, as a way of demonstrating the ARM binary images compiled using our cross toolchain, actually execute.

Download the GDB source:

```
$ wget http://mirror.anl.gov/pub/gnu/gdb/\
>gdb-7.5.tar.gz
```

Extract the source code:

```
$ cd $BASE
$ tar zxvf gdb-7.5.tar.gz
```

Configure GDB:

```
$ cd gdb-7.5
$ ./configure --target=$TARGET --prefix=$TOOLCHAIN
```

Compile and install;

```
$ make && make install
```

Set the PATH and LD_LIBRARY_PATH environment variables:

```
$ export PATH=$PATH:$TOOLCHAIN/bin
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$TOOLCHAIN/lib
```

Compile the code in Listing 6.1 with debugging option set (`-g` option):

```
$ arm-elf-gcc -g -o hello hello.c
```

We can see that the resultant executable file *hello* for the ARM platform:

```
$ file hello
hello: ELF 32-bit LSB executable, ARM, version 1,
statically linked, not stripped
```

Run GDB. This gives us the `(gdb)` prompt:

```
$ arm-elf-gdb hello
GNU gdb (GDB) 7.5
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and
redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type"show copying"
and "show warranty" for details.
This GDB was configured as "--host=i686-pc-linux-gnu
--target=arm-elf".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/aholt/arm/
hello...done.
(gdb)
```

Start the ARMulator by setting the target to simulator:

```
(gdb) target sim
Connected to the simulator.
```

Make the program available for debugging:

```
(gdb) load
Loading section .init, size 0x20 vma 0x8000
Loading section .text, size 0x359c vma 0x8020
Loading section .fini, size 0x1c vma 0xb5bc
Loading section .rodata, size 0x1c vma 0xb5d8
Loading section .eh_frame, size 0x4 vma 0xb5f4
Loading section .ctors, size 0x8 vma 0x135f8
Loading section .dtors, size 0x8 vma 0x13600
Loading section .jcr, size 0x4 vma 0x13608
Loading section .data, size 0x954 vma 0x1360c
Start address 0x810c
Transfer rate: 129792 bits in <1 sec.
```

Run the hello world program:

```
(gdb) run
Starting program: /home/aholt/arm/hello
hello world
[Inferior 1 (process 42000) exited normally]
```

As we can see from the output above, the string “hello world” is displayed demonstrating the successful execution of the ARM binary. To quit GDB, use:

```
(gdb) quit
```

6.5 Summary

In this chapter we have introduced the topic of compiler toolchains. We have, however, avoided an academic treatment of the subject, preferring to present a number of worked examples instead. As the topic of this book is embedded operating systems, we are interested in cross compiler toolchains where the host upon which embedded applications are built are not necessarily the same architecture as the target system.

We showed how to build a basic cross compiler toolchain for an ARM architecture. We tested it using an ARM emulator that runs on the GDB debugger.

Embedded ARM Devices

7

ARM is a family of RISC (reduced instruction set computing) processors that are used extensively in the mobile device market. ARM Holdings plc designs the ARM family of processors but the processors themselves are manufactured by other companies under license. The ARM processor was originally developed by Acorn Computers in collaboration with Apple. Although the acronym, "ARM" stands for advanced RISC machine, it originally stood for *Acorn* RISC Machine. Acorn were the company that developed the BBC Micro which was commissioned for The Computer Programme as part of the BBC's "Computer Literacy Project". The BBC Micro was highly successful and was used widely in UK schools. In this chapter we look at two embedded devices based upon the ARM processor, namely, the Raspberry Pi and the BeagleBone.

The Raspberry Pi [1,2] was designed to be an inexpensive personal computer (sub £30). Despite its small size, the device has a powerful graphics processor, USB ports for mouse and keyboard and an HDMI port to connect a monitor. Like the BBC Micro, the Raspberry Pi was conceived for educational purposes. It is no coincidence that there are A and B versions of the Raspberry Pi just as there were for the BBC Micro.

There are a number of GNU/Linux distributions for the Raspberry Pi. The main one is Raspbian (based upon Debian). Raspbian, however, is a distribution aimed at the desktop market and comes with a graphical user interface. It is not necessarily an embedded operating system. Arch Linux, in contrast, is more suited embedded systems.

BeagleBones are part of the BeagleBoard series of low-power open source hardware, single-board computers. The BeagleBones are produced by Texas Instruments and Digi-Key [3] and are aimed at the educational and hobbyist market. There are two versions of the BeagleBone called the BeagleBone and the BeagleBone Black. The BeagleBone is shipped with the Angstorm GNU/Linux distribution, but can also run Android and Ubuntu (at the time of writing this book).

Both devices incorporate expansion headers on the board in order to access the general purpose input/output (GPIO) pins. GPIO pins are used in embedded systems to transmit and receive digital signals between peripheral devices.

A. Holt and C.-Y. Huang, *Embedded Operating Systems*, Undergraduate Topics in Computer Science, DOI 10.1007/978-1-4471-6603-0_7

The Raspberry Pi board has a single 26 pin (2 × 13 arrangement) expansion header. The BeagleBones has two, 46 pin (in a 2 × 23 arrangement) expansion headers labelled P8 and P9 on the board. As well as general input/output, the GPIO pins can be programmed to provide the following interfaces:

- Inter-integrated Circuit (I^2C)
- Serial peripheral interface (SPI)
- Universal asynchronous receiver/transmitter (UART)

Pins for +3.3 V, +5 V and GND supply lines are also available from expansion headers.

This chapter is organised into two sections, one for each system. We show how to get started with each system and how to access them. We also take the opportunity to explore the physical computing features they provide.

7.1 Raspberry Pi

The Raspberry Pi is a small form factor desktop PC but is also suited to embedded applications. The board features a Broadcom BCM2835 SoC (system on a chip) with a ARM1176JZFS 700 MHz processor. It also includes a Videocore 4 GPU which is capable of BluRay quality playback. There are two models of Raspberry Pi, model A and model B. Figure 7.1 shows a photograph of the two models. Table 7.1 compares their specifications.

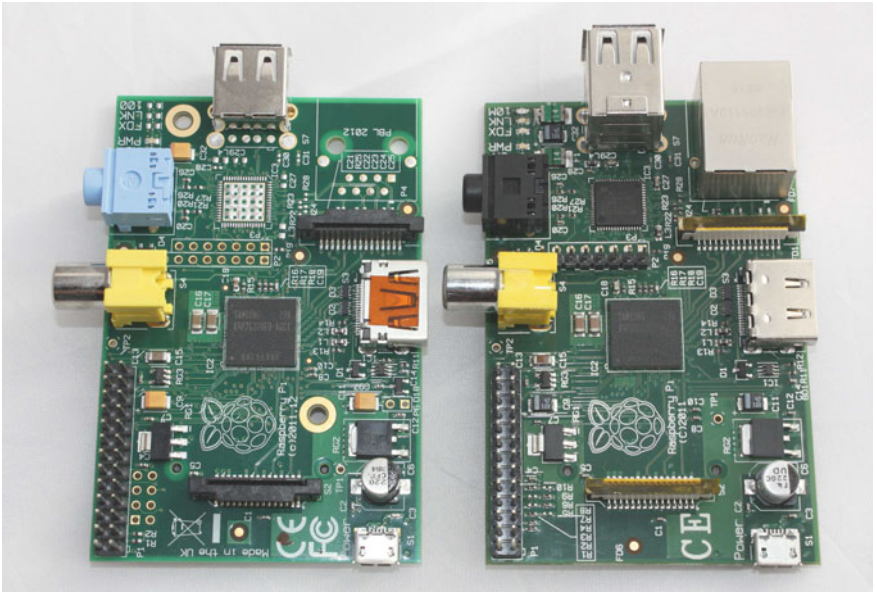


Fig. 7.1 Raspberry Pi. Model A (*left*), model B (*right*)

Table 7.1 Raspberry Pi models

Model	Architecture	Speed	Memory	Ethernet	USB
A	ARM11	700 MHz	256 MB	No	1
B	ARM11	700 MHz	512 MB	Yes	2

7.1.1 Installing an Operating System

At the time of writing this book, the Raspberry Pi supports a number of operating systems:

- Raspbian: Debian for Raspberry Pi.
- Arch Linux: Arch Linux is a simple, lightweight GNU/Linux distribution.
- Android: Google's smart phone and tablet OS.
- Risc OS. RISC OS [4,5] was designed by Acorn Computers Ltd to run on the ARM chipset.
- Plan 9: Plan 9 from the Bell Labs [6] is an operating developed by the Bell Laboratories as a successor to Unix. It is primarily for researchers and hobbyists.

For the purpose of this example, we will focus on Raspbian. Download the latest version of Raspbian which, at the time of writing this book, is Wheezy:

```
$ wget http://downloads.raspberrypi.org/raspbian_latest
```

Uncompress the image file:

```
$ unzip 2013-05-25-wheezy-raspbian.zip
Archive:  2013-05-25-wheezy-raspbian.zip
  inflating: 2013-05-25-wheezy-raspbian.img
```

Connect the SD card to the PC (you may need an SD card reader) and unmount any filesystems that automatically mounted. Write the image to the SD card:

```
$ sudo dd if=2013-05-25-wheezy-raspbian.img \
> of=/dev/sdb bs=1M
850+0 records in
1850+0 records out
1939865600 bytes (1.9 GB) copied, 509.074 s, 3.8\,MB/s
```

Remove the SD card from the host PC and insert into the SD slot of the Raspberry Pi. When the Raspberry Pi is powered on the operating system will boot.

7.1.2 Using the Raspberry Pi Serial Port

In this subsection, we present a simple exercise with the Raspberry Pi's GPIO ports. We show how to connect to the Raspberry Pi through its serial port. This is useful if unit is a model A. The model A lacks an Ethernet port which prevents remote access to the command-line[1] and would otherwise need a monitor, keyboard and mouse. The aim of this exercise is to show how the Raspberry Pi's serial port can be accessed using a USB serial port of a PC.

The serial port is accessible through the GPIO ports 14 and 15, which are the serial transmit (TXD) and receive (RXD) respectively. There are a number of ways of connecting a serial USB to the Raspberry Pi's GPIO ports. Probably the easiest way is to use a console lead. The console lead in [7] has female connectors which can be plugged onto the GPIO pins directly. However, we use the Arduino Uno as an *interface* between the USB interface and the GPIO pins. A list of components used are given below:

- Raspberry Pi. In this exercise we used a Model A, but a Model B can also be used. The voltages on GPIO pins should not exceed 3.3 V, otherwise the Raspberry Pi could be damaged. For this reason, care should be taken when connecting to the GPIOs. As we found out the hard way, another reason for using a Model A for this exercise, is it is less expensive to destroy than a Model B.
- Slice of Pi. Break out board for the Raspberry Pi [8]. We use the break out board as a convenient way to connect to the GPIO pins. Note that, the pin headers need to be soldered to the PCB.
- Arduino. We used an Arduino Uno as it has pin headers for the GPIOs.
- 2× USB cables:
  - Type-A to type-B cable USB for communication between the PC and Arduino. This also provides power to the Arduino.
  - Type-A to type-B Micro to provide power to the Raspberry Pi.
- An assortment of jumper cables.

We need to download a blank sketch to the Arduino so that the bootloader does not respond to activity on the USB serial interface. Listing 7.1 shows the code for the sketch.

Listing 7.1 *Blank sketch for the Arduino*

```
void setup() {
}
void loop() {
}
```

[1] Although, we could purchase 802.11n USB adapter and use a WiFi network.

We need to install the Arduino IDE on our host PC. Download the latest Adrduino IDE (at the time of writing this book, the latest release is 1.5.4).

```
$ cd arduino-1.5.4
$ ./arduino
```

Connect the Arduino to the host PC using the appropriate USB cable. Run the dmesg command on the host, you should see a line like this:

```
[ 6094.178626] cdc_acm 5-1:1.0: ttyACM0: USB ACM device
```

This tells us the Arduino is connected to the */dev/ttyACM0* device. Run the Arduino IDE and enter the code in Listing 7.1. Figure 7.2 shows the IDE with sketch code entered. Ensure that the IDE is connected to the correct serial devices. (*/dev/ttyACM0* in our case). Click on the Tools tab and from the drop down menu, select the Serial Port menu item. Select the serial device on which the Arduino is connected. Next,

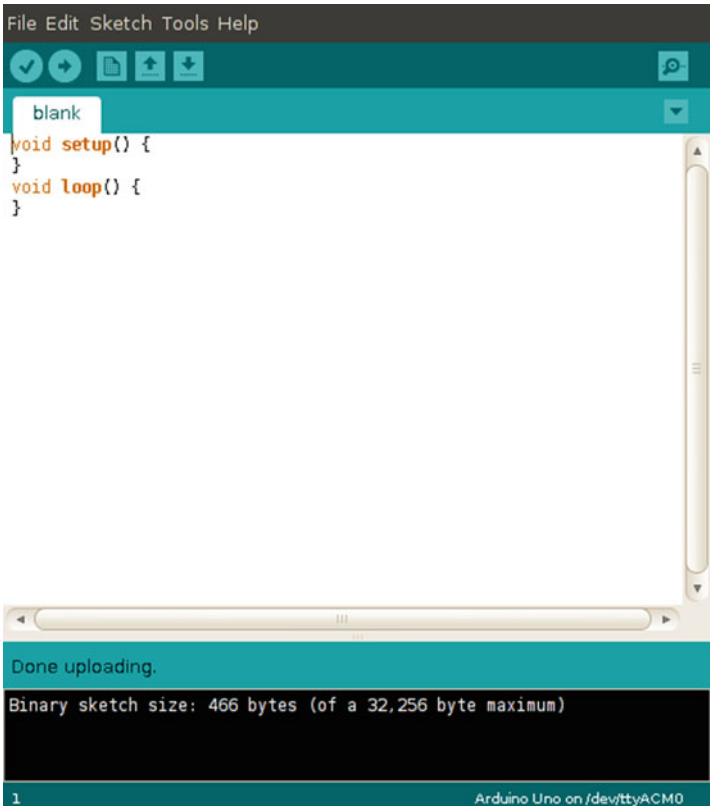


Fig. 7.2 Arduino IDE

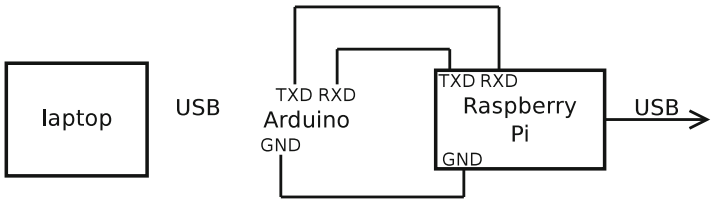


Fig. 7.3 Raspberry Pi serial connection

we set the Arduino board type. From Tools drop down menu, select the Board menu option. From here, configure the IDE for the appropriate Arduino (in our case, this is an Arduino Uno).

Compile the sketch by clicking on the button marked with a "✓". If the compilation is successful, download the compiled sketch to the Arduino by clicking on the "→" button. The IDE can now be closed down.

With the Raspberry Pi turned off (USB disconnected) attach the Slice of Pi break out board. Then connect the RX pin on the Arduino (digital pin 0) to RX on the Raspberry Pi. Similarly connect the Arduino TX pin (digital pin 1) to the Raspberry Pi TX pin. Finally connect the ground (GND) pins of both units. The diagram in Fig. 7.3 shows a schematic of the connection between the Arduino and Raspberry Pi (but at this stage, the USB to the Raspberry Pi should not be connected). On the host PC, run a terminal emulator:

```
$ screen /dev/ttyACM0 115200
```

Power up the Raspberry Pi by connecting the USB cable. Kernel messages should start streaming to the screen as device boot up. After the start up scripts have run, a login prompt should appear. Login with the user account name "pi" and default password "raspberry":

```
raspberrypi login: pi
Password: raspberry
pi@raspberrypi:~$
```

If this is the first time that the Raspberry pi has been booted, then a message will be displayed stating the configuration is incomplete. Run the configuration utility below:

```
$ sudo raspi-config
```

This displays a menu of options for customising the Raspberry Pi. We recommend you select option 1, so the filesystem size is expanded to the full size of the SD card.

Secure shell (SSH) can be enabled/disabled through the Advanced options (option 8). In our case we are using a model A which does not have an Ethernet

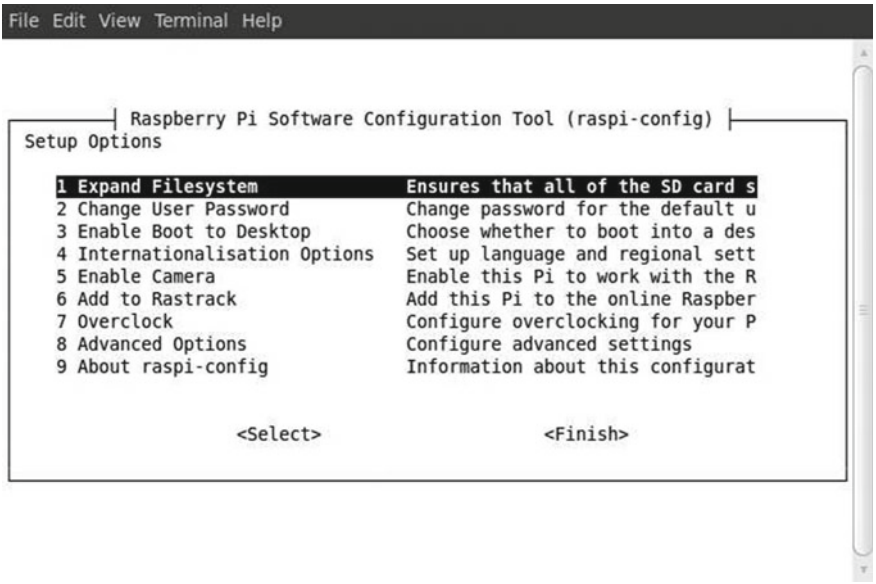


Fig. 7.4 Raspberry Pi initial configuration menu

port. For this reason we leave SSH disabled. If you have a model B (which has an Ethernet port) or you have a USB WiFi device (which can be used with a model A) then you may wish to enable SSH (Fig. 7.4).

The Raspberry Pi uses NTP (network time protocol) by default to sync its clock. As our device lacks a network connection we found the time was several months out. We, therefore, set the date manually:

```
pi@raspberrypi:@$ sudo date -s"27 JUN 2013 14:33:00"
Thu Jun 27 14:33:00 BST 2013
```

We also disabled the NTP daemon (if you have a Raspberry Pi with network capabilities we suggest you omit these steps). Make the directory for runlevel 2 scripts the current directory:

```
pi@raspberrypi:@$ cd /etc/rc2.d
```

In this directory there should be a symbolic link to the NTP start-up script in */etc/init.d/*:

```
pi@raspberrypi:/etc/rc2.d$ ls *ntp
S02ntp
```

Change the name of the symbolic link so that it begins with a "K":

```
pi@raspberrypi:/etc/rc2.d$ sudo mv S02ntp K02ntp
```

Run the update-rc.d script, but ignore any errors:

```
pi@raspberrypi:/etc/rc2.d$i sudo update-rc.d script defaults
update-rc.d: using dependency based boot sequencing
update-rc.d: error: unable to read /etc/init.d/script
```

Reboot the unit:

```
$ sudo shutdown -r now
```

7.1.3 Remote Serial Port

In this subsection we extend the serial port exercise in Sect. 7.1.2 so that the serial port can be used remotely over a wireless *ZigBee* link. ZigBee is a low power wireless protocol for personal area networks (PANs) based upon the IEEE 802.15.4 standard. Given that transmission power is low, ZigBee is designed for applications with low data rate requirements over short distances. The ZigBee protocol should, therefore, be well suited for accessing the Raspberry Pi's serial console remotely. The items of equpiment needed for this exercise are given Below.

- Raspberry Pi (Model A or B). In our case, we use Model A.
- Slice of Pi breakout board.
- 2 × ZigBee transceivers (XBee modules).
- USB Serial to XBee adapter.

The diagram in Fig 7.5 shows how the component parts are connected. Note, the dotted line represents a wireless link.

The Slice of Pi breakout board connect the the expansion headers of the Rapsberry Pi. There are expandsion headers on the Slice of Pi for the XBee module. Figure 7.6 shows how the XBee module interfaces to the Raspberry Pi using the Slice of Pi breakout board.

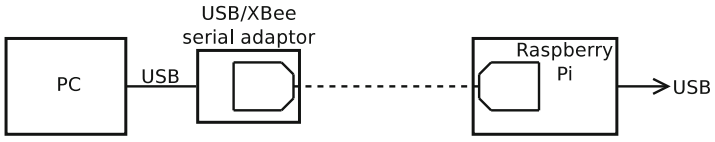


Fig. 7.5 Raspberry Pi remote serial connection over ZigBee

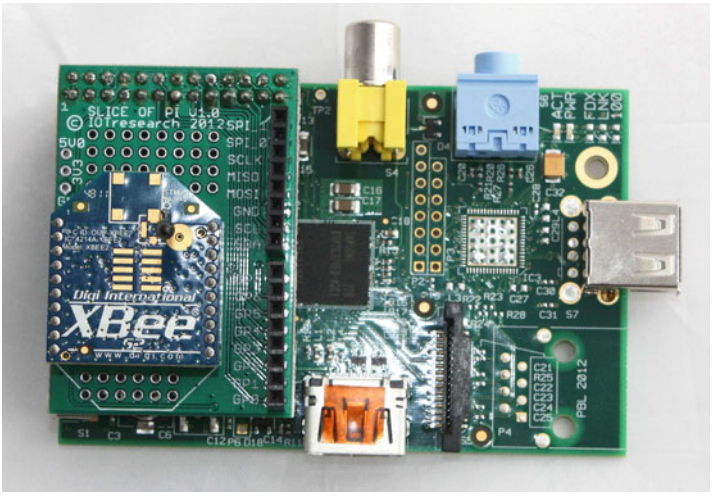


Fig. 7.6 XBee module attached to Raspberry Pi

Configuration of the XBee modules is performed using a utility called X-CTU (which is free to download). Unfortunately, there is only a Windows version of the X-CTU available. However, it is possible to run X-CTU on a GNU/Linux system under Wine (Wine is not an emulator). Wine can be installed using the package management system. On Debian and derivatives, use:

```
$ sudo apt-get install wine
```

Download the X-CTU installer utility:

```
$ wget http://ftp1.digi.com/support/utilities/\
> 40003002_B.exe
```

As X-CTU is a Microsoft Windows application, it used Windows nomenclature to specify the devices serial ports, namely, COM1, COM2 etc., whereas GNU/Linux uses file object pathnames to reference devices. In the *dosdrives* directory, create a number of symbolic links the serial devices we are likely use:

```
$ cd ~/.wine/dosdevices
$ ln -s /dev/ttyUSB0 com10
$ ln -s /dev/ttyUSB1 com11
$ ln -s /dev/ttyACM0 com12
```

We need to configure the two XBee modules for different roles. One module is configured as the *coordinator* and the other as a *router*. Each module has a unique 64 bit serial number which we will use in the configuration procedure. The serial

Table 7.2 XBee serial numbers

Role	32-msb	32-lsb
Coordinator	0013A200	408A8966
Router	0013A200	408A8957

numbers are printed on the underside of the module. The 32 most significant bits (msb) of the serial number for XBee modules is 0013A200. The 32 least significant bits (lsb) are unique to the module.

Choose one module to be configured as the coordinator and one to be configured as the router (the choice is arbitrary). Make a note of the respective serial numbers. Table 7.2 shows the serial numbers for the coordinator and router of our modules (the serial number for your modules will be different). When we configure the modules, we use the serial number of the router as the destination address in the coordinator and vice versa. This will become apparent when we describe the configuration process but you should ensure you have these serial numbers to hand when you perform it.

We will begin with the coordinator. Start the X-CTU utility. A window like the one in Fig. 7.7 will be displayed on the screen.

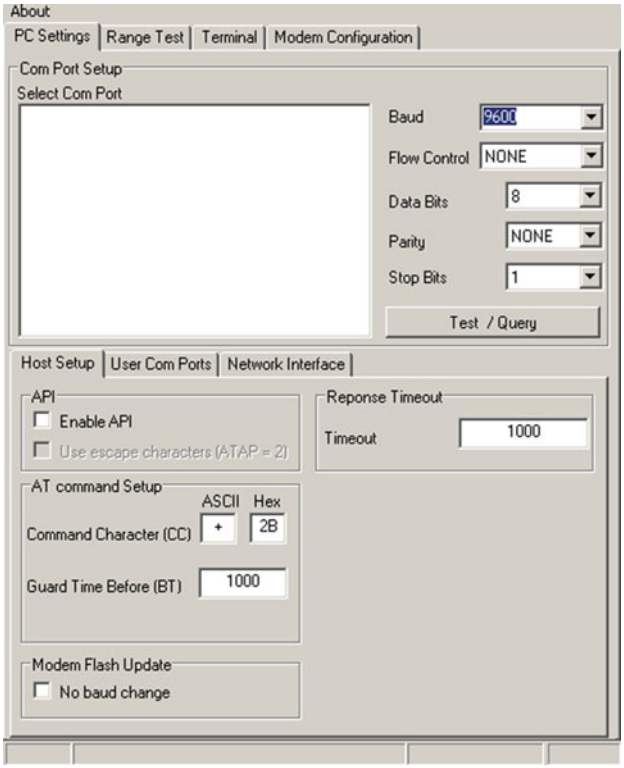


Fig. 7.7 X-CTU

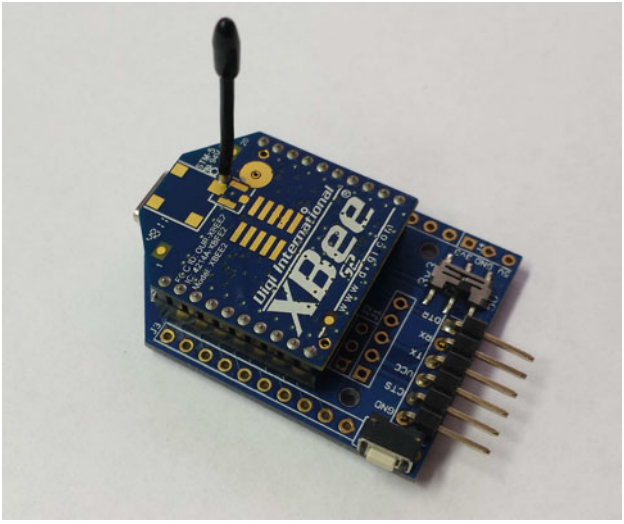


Fig. 7.8 The XBee module connected to USB serial converter

We must first establish communication with the XBee module. Connect the host to the XBee module using the USB serial converter [9] (and USB cable). Figure 7.8 shows the XBee module seated in the expansion headers of the USB to serial converter.

Check dmesg to see which serial port the XBee module attaches to. In our case, it is */dev/ttyUSB0* which we defined as *com10*. Select the User Com Ports tab in X-CTU (Fig. 7.9). In the Com Port Number text box, enter "COM10" (characters are displayed in upper case regardless of the case they are entered) and click the Add button. "COM10" will appear as an option in the Select Com Port box.

Click on COM10 in the Select Com Port box to highlight it. Then select the Test/Query button. If the test is successful, you should see a message window similar to one in Fig. 7.9.

- Check that pins are correctly seated in the sockets. It is easy to insert the XBee module with the pins misaligned in the sockets.
- Check the XBee unit is connected correctly. Did a message appear in dmesg?
- Check the serial port is correct (dmesg).
- Check the symbolic links in *dosdrives* point to the correct interfaces?
- Check the communication parameters are correct. The default communication parameters for the XBee modules are 9600 baud, 8-bits, no parity. As we can see from Fig. 7.9, the settings in X-CTU reflect the default settings. If the parameters on the XBee module have been changed then set them accordingly.

As well as confirming communication between the PC and XBee modules, the Test/Query Modem dialogue box also gives us other information about the module, namely:

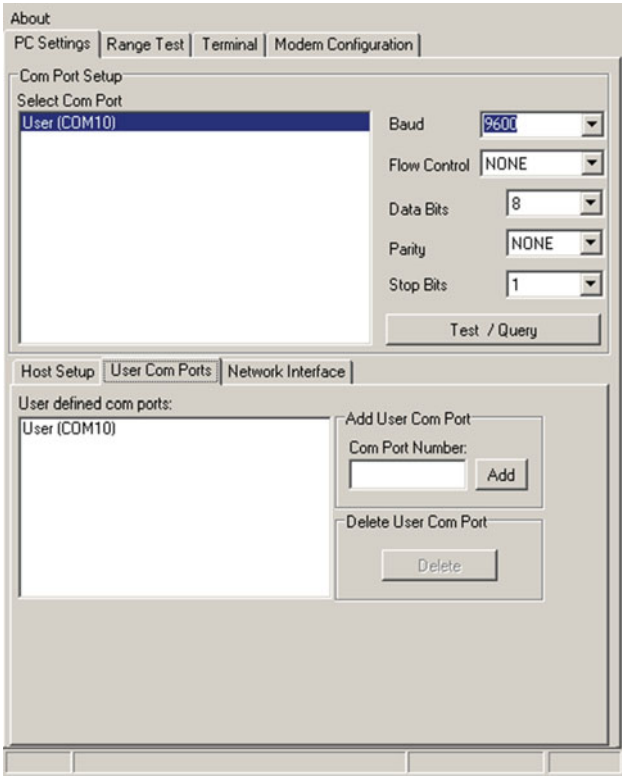


Fig. 7.9 X-CTU

- Modem type: XB24-B
- Modem firmware version: 1020
- Serial Number: 13A200408A8966

Select the Modem Configuration tab. Click the Read button. X-CTU will read the parameters from the module (Fig. 7.10).

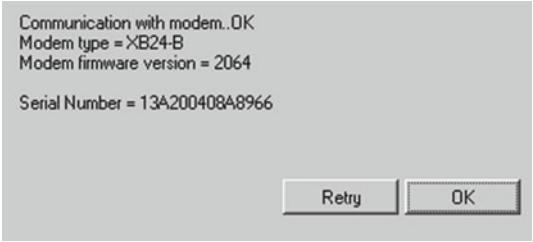


Fig. 7.10 X-CTU modem query

Modem Type The modem type for our module is XB24-B but we can upgrade it to the XB24-ZB firmware. Click on the down arrow of the Modem XBee and select "XB24-ZB" from the drop down menu.
Function Set Configure this module as a coordinator. Under Function Set, bring up the drop down menu and select "ZIGBEE COORDINATOR AT".
Baud rate The baud rate of the module needs to match that of the Raspberry Pi serial port, which is 115200 baud. In the panel with the scroll bar, scroll down to the Serial Interface section. There is a parameter for the Baud Rate. From the drop down menu for Baud Rate, select 115200.
Destination address The destination address is a serial number of the *other* router. As we are configuring the coordinator here, these fields must contain the serial number of the router. The destination address is divided into two components, namely, DH and DL. In the DH field enter the most significant 128 bits of the router's serial number: 13A200 (the leading zero can be omitted). In the DL field, enter the least significant bits of the router's serial number. In our case, this is 408A8957 (see Table 7.2).

On completion of the step above, click the Write button to download the parameters to module. As we are upgrading the firmware, a dialogue box is displayed similar to the one shown in Fig. 7.11. Respond to this dialogue box by pressing and releasing the reset button on the module. The firmware download will then resume.

On successful completion of the firmware download, we recommend returning to the PC Settings tab to run a Test/Query on the module. Ensure you change the Baud setting to 115200. This completes the configuration of the coordinator. The procedure for configuring the router/end device module is similar to that of the coordinator. Just as with the coordinator set Modem XBee to "XB24-ZB" and baud rate to 115200.

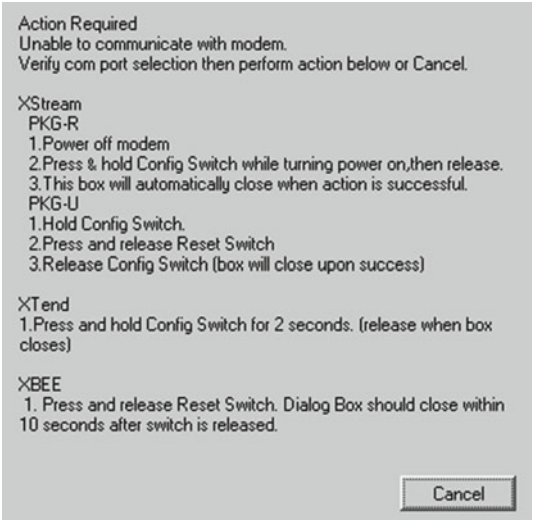


Fig. 7.11 X-CTU info

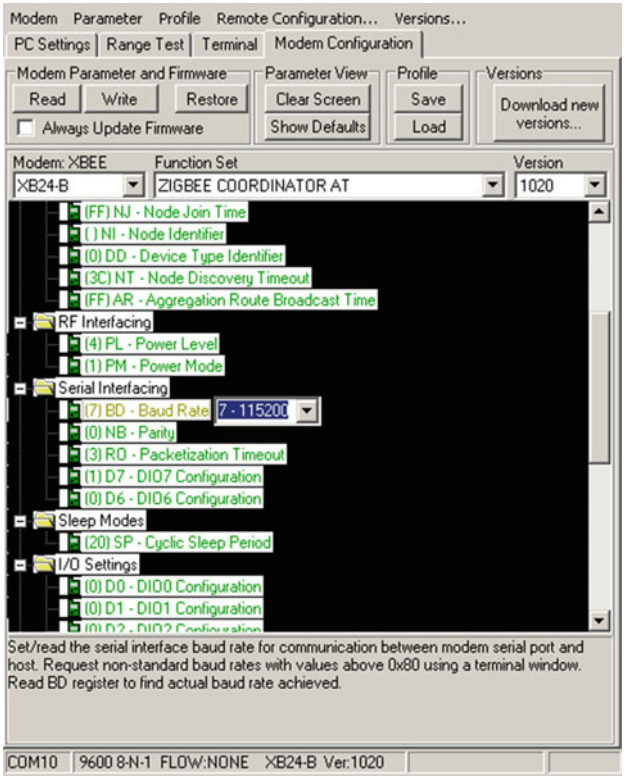


Fig. 7.12 Configure XBee

However, for the Function Set, select "ZIGBEE ROUTER DEVICE AT". Set DH and DL (destination address) to 0013A200 and 408A8966 respectively (where, for the DL field you should substitute the 32-lsb for your XBee router module. Click the Write button to download the firmware to module (Fig. 7.12).

Power off the Raspberry Pi, then connect one of the XBee modules to it using the break out box. Either module can be used here, but used the XBee module configured as the router. Connect the other XBee module (in our case the coordinator) to the PC with the XBee adapter and the (type) USB cable. Run the terminal emulator:

```
$ screen /dev/ttyUSB0 115200
```

A preliminary test of the XBee module can be performed by typing `+++`. An `OK` message should be displayed on the screen. This places the module into command mode. The modules understand AT-like commands just like regular modem. We can query the destination address with the ATDH and ATDL commands which display the respective 32-msb and 32-lsb of the destination address:

```
atdh
13A200
atdl
408A8957
```

This confirms the destination address of the coordinator is that of the router. You should note, that after a few seconds of inactivity, the mode will revert back to transparent. If you wish to enter mode AT commands, you will need to type +++ again.

Power up the Raspberry Pi. After a few seconds the kernel messages will be displayed on the screen. Once the boot process has completed the login prompt will appear.

```
Debian GNU/Linux 7.0 raspberrypi ttyAMA0
raspberrypi login:
```

7.2 BeagleBone

In this section we show how to set up both the BeagleBone and BeagleBone Black (see Fig. 7.13). We also give a brief introduction into physical computing programming on BeagleBones. BeagelBones are *bare bones* versions of the BeagleBoard series. Their

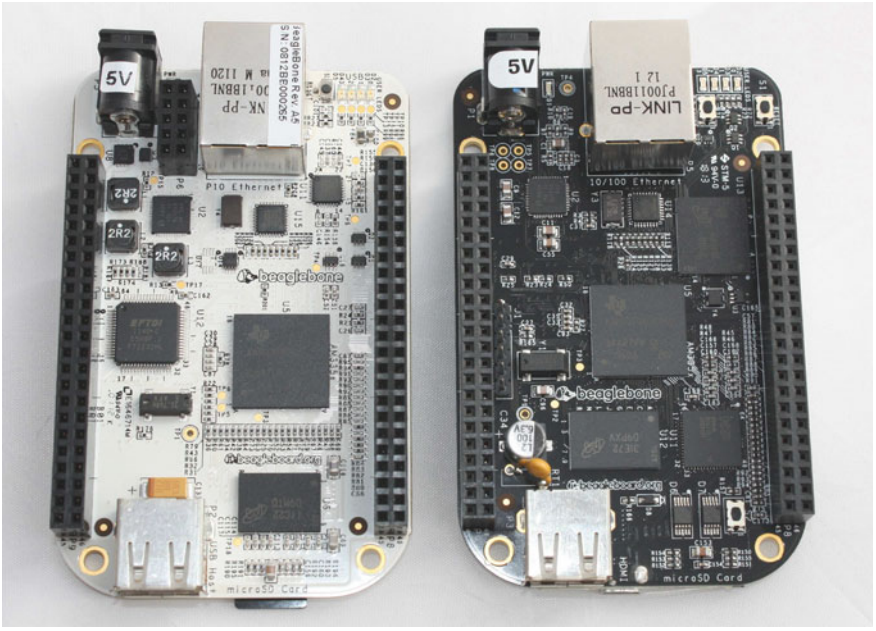


Fig. 7.13 Original BeagleBone (*left*) and BeagleBone *Black* (*right*)

Table 7.3 BeagleBoards technical specification

Board	Processor	Speed	Memory (MB)
BeagleBoard	OMAP3530 ARM Cortex-A8	720 MHz	256
BeagleBoard-xM	AM37x ARM Cortex-A8 compatible	1 GHz	512
BeagleBone	AM335x ARM Cortex-A8	720 MHz	256
BeagleBone black	AM335x ARM Cortex-A8	1 GHz	512

technical specification is shown in Table 7.3 (the table also shows the specification of the other BeagleBoards, though we do not cover them in this book).

7.2.1 Setting Up the BeagleBone

In this subsection we describe how to get the BeagleBone devices working. In order to avoid confusion, we use the terms *original BeagleBone* and *BeagleBone Black* to distinguish between the respective models. The term *BeagleBone* refers to either model when the distinction is unimportant. There are small differences between the set up of the original BeagleBone and the BeagleBone Black which we will cover in this section.

Connect the BeagleBone to a personal computer (running GNU/Linux) using a USB cable. Run the dmesg command. If you are using an original BeagleBone, then you will see a line similar to the one below:

```
[10791.561172] usb 2-1.1: FTDI USB Serial Device
converter now attached to ttyUSB0
```

This tells us that the BeagleBone is connected to the */dev/ttyUSB0* serial device. For BeagleBone Black, however, the host uses a different device driver. The line in the dmesg results shows that the host is connected to the BeagleBone Black using */dev/ttyACM0*:

```
[ 7944.956365] cdc_acm 2-1:1.2: ttyACM0: USB ACM device
```

Use the command below to connect to the BeagleBone's serial console, ensuring that you use the appropriate console device:

```
$ screen /dev/ttyUSB0 115200
```

A banner and the login prompt is displayed. In this example, we are using an original BeagleBone running the Angstrom distribution it came with:

```
.---O---.
|       |                  .-.           o o
|   |   |-----.-----.-----.| |   .----..-----.-----.
|       |     | __  |  ---'|'--.|  .-'|     |     |
|   |   |  |  |     |---  ||  --'|  |  | '  | | | |
'---'---'--'--'--.  |-----''----''--' '-----'-'-'-'
                -'  |
               '---'

The Angstrom Distribution beaglebone ttyO0

Angstrom v2012.01-core - Kernel 3.2.5+

beaglebone login: root
Last login: Thu Mar 15 15:21:52 UTC 2012 on ttyO0
root@beaglebone:~#
```

At this point, we recommend you give the root account a password (using the passwd command). As well as providing a serial over the USB interface the BeagleBone comes up as a mass storage device on the host PC:

```
$ mount
/dev/sdb on /media/BEAGLE_BONE type vfat (rw,nosuid,
nodev,uhelper=udisks,uid=1000,gid=1000,shortname=
mixed,dmask=0077,utf8=1,flush)
```

This USB interface can also be used for IP networking. The difference between the two models is that the BeagleBone Black can support both services simultaneously whereas the original BeagleBone can only support one or other. If we look at the modules on the original BeagleBone, we see the `g_mass_storage` module is currently loaded:

```
root@beaglebone:~# lsmod
Module                  Size  Used by
g_mass_storage         24010  0
ipv6                  210434  16
```

In order to establish networking between the host PC and the original BeagleBone we need to eject the storage device on the host:

```
$ sudo eject /dev/sdb
```

It is important to note that, it is not necessary to eject the storage device with the BeagleBone Black, the network interfaces will be brought up automatically

when the device boots.Returning to the original BeagleBone, we see that the `g_mass_storage` has been replaced by the `g_ether` module:

```
root@beaglebone:~# lsmod
Module                  Size  Used by
g_ether                27775  0
ipv6                  210434  16
```

This has the effect of bringing up a `usb0` Ethernet interface on the original BeagleBone (on the BeagleBone Black the `usb0` will come up automatically):

```
root@beaglebone:~# ifconfig usb0
usb0      Link encap:Ethernet  HWaddr BE:4C:75:39:6E:08
          inet addr:192.168.7.2  Bcast:192.168.7.3
Mask:255.255.255.252
          inet6 addr: fe80::bc4c:75ff:fe39:6e08/64
Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500
Metric:1
          RX packets:19 errors:0 dropped:0 overruns:0
frame:0
          TX packets:30 errors:0 dropped:0 overruns:0
carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:5214 (5.0 KiB)  TX bytes:5748
(5.6 KiB)
```

A corresponding Ethernet interface is brought up on the host PC:

```
$ ifconfig eth2
eth2      Link encap:Ethernet  HWaddr d4:94:a1:8b:39:99
          inet addr:192.168.7.1  Bcast:192.168.7.3
Mask:255.255.255.252
          inet6 addr: fe80::d694:a1ff:fe8b:3999/64
Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500
Metric:1
          RX packets:30 errors:0 dropped:0 overruns:0
frame:0
          TX packets:19 errors:0 dropped:0 overruns:0
carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:5328 (5.3 KB)  TX bytes:5480 (5.4 KB)
```

From the BeagleBone (original or Black edition), we can verify that there is a network connection between the target and the host with the Ping command:

```
root@beaglebone:~# ping -c 4 -q 192.168.7.1
PING 192.168.7.1 (192.168.7.1) 56(84) bytes of data.

--- 192.168.7.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss,
time 2998ms
rtt min/avg/max/mdev = 0.233/0.290/0.338/0.041 ms
```

Attempts to connect to systems beyond the host, however, will fail:

```
root@beaglebone:~# ping -c 4 -q 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.

--- 8.8.8.8 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss,
time 3001ms
```

This is because the network infrastructure the host is connected to, does not know how to route to the network 192.168.7.0/24. This is a similar problem to one we encountered with UML virtual machines in Chap. 4. We can resolve the issue by implementing NAT (network address translation) on the host. Enable IP packet forwarding so that the host can forward packets between usb0 and its Ethernet port (in our case eth0):

```
$ sudo -i
# echo 1 > /proc/sys/net/ipv4/ip_forward
# exit
logout
```

On the host, configure NAT using the following sequences of iptables commands:

```
$ iptables -F
$ iptables -A INPUT -i lo -j ACCEPT
$ iptables -A POSTROUTING -t nat -o eth0 \
> -s 192.168.7.0/24 -d 0/0  -j MASQUERADE
$ iptables -A FORWARD -t filter -o eth0 -m state \
> --state NEW,ESTABLISHED,RELATED -j ACCEPT
$ iptables -A FORWARD -t filter -i eth2 -m state \
> --state ESTABLISHED,RELATED -j ACCEPT
```

For an explanation of these commands see Sect. 4.5. The BeagleBone can now communicate with the external network. We should emphasise, networking using the

USB interface is just one option. Both the BeagelBones have regular Ethernet interfaces which can be configured for IP.

7.2.2 Physical Computer Programming on the BeagleBone

The BeagleBones have a number of GPIO (general purpose input/output) pins and LEDs, which are programmable. To illustrate physical computer programming on the BeagleBone we present a simple example on how to interact with the four USER LEDs. The USER LEDs are located near the S3 power switch and are labelled USER 1 through to 4 on the board. The LEDs (and GPIOs) can be controlled by writing to files in sysfs. For brevity define the pathname to the directory for the BeagleBone's LEDs:

```
$ LEDSPATH="/sys/devices/platform/leds-gpio/leds/
```

If we examine the contents of the $LEDSPATH directory, we see a subdirectory for each of the LEDs:

```
root@beaglebone:~# ls ${LEDSPATH}
beaglebone::usr0  beaglebone::usr1  beaglebone::usr2
beaglebone::usr3
```

Notice, that directories are labelled 0 to 3 and correspond to USER LEDs 1 to 4 respectively. These directories contain files, subdirectories and symbolic links which relate to the individual LEDs. Choosing, USER LED 4, for example:

```
root@beaglebone:~# ls ${LEDPATH}/beaglebone::usr3
brightness  device  max_brightness  power  subsystem
trigger  uevent
```

We can turn LEDs on and off by changing the content of the *brightness* file. For example, USER LED 4 can be turned on by writing a "1" to the corresponding *brightness* files:

```
echo $1 > ${GPIOPATH}/beaglebone\:\:usr3/brightness
```

Conversely, writing a "0" to the corresponding *brightness* file turns LED 4 off:

```
echo $1 > ${GPIOPATH}/beaglebone\:\:usr3/brightness
```

The Python code in Listing 7.2 shows a script for turning the LEDs on then off in a repeated sequence. It defines an UserLED class that, when declared, initialises the LEDs (turns them off). The UserLED class supports methods for turning the LEDs both on and off by writing to LED the *brightness* files in sysfs.

Listing 7.2 *Contents of led_seq.py*

```
#!/usr/bin/env python

import sys,time

ledpath ="/sys/devices/platform/leds-gpio/leds"

class UserLED():

    def __init__(self, n):
        self.led ="%s/beaglebone::usr%d/brightness"% \
                        (ledpath,n)
        self.clear()

    def set(self):
        open(self.led,'w').write("1")

    def clear(self):
        open(self.led,'w').write("0")


def led_seq():
    for method in ["set","clear"]:
        for i in xrange(4):
            led ="USR%d.%s()"% (i,method)

if __name__=='__main__':

    # Declare LED objects
    USR0 = UserLED(0)
    USR1 = UserLED(1)
    USR2 = UserLED(2)
    USR3 = UserLED(3)

    while True:
        led_seq()
```

Create a file *led_seq.py* with the contents of Listing 7.2. Give the file execute permissions:

```
root@beaglebone:~# chmod +x led_seq.py
```

Run the script:

```
root@beaglebone:~# ./led_seq.py
```

The USER LEDs will blink on and off in a repeated sequence. Now we show how to program the BeagleBone using the system's IDE (integrated development environment). The BeagleBone and BeagleBone Black both come with an IDE called Cloud9. We demonstrate how to use the Cloud9 IDE to program the physical hardware of the BeagleBone.

The programming language supported by Cloud9 is JavaScript. It comes with the Bonescript library which provides an API to control the physical hardware (GPIOs and LEDs). Bonescript is an optimised Node.js library for BeagleBoards. Node.js was designed for developing scalable network applications by using non-blocking, event-driven I/O. I/O requests to the BeagleBone's physical hardware are implemented as asynchronous *callbacks*. An event loop runs and executes the corresponding callback function for each event as it occurs. The advantage of this programming method, is that the program does not block I/O requests.

Cloud9 can be accessed using a Web browser by entering the URL: 192.168.7.2:3000. When using Cloud9 for the first time, a dialogue box is displayed from which you can select a detailed guided tour of the IDE. It is a good idea to take the guided tour at some point, but for now, just click the button "Just the editor, please". Figure 7.14 shows the Cloud9 IDE running in a Web browser.

The program in Listing 7.3 shows the JavaScript code to interact with the four USER LEDs on the board. This will control the LEDs in the same way as the Python script in Listing 7.2. Create a new project by clicking on the + in the projects tab. Enter the code in Listing 7.3 into the new tab (see Fig. 7.15). Then save the project to a file by pressing CONTROL-S. You will be prompted to give the file a name. In this example, we call it *led_seq.js* and save it in the *demo* directory. Execute the program by clicking on the "run" icon in the top menu bar. The "run" icon is replaced by a "stop" icon (which can be used to terminate the program).

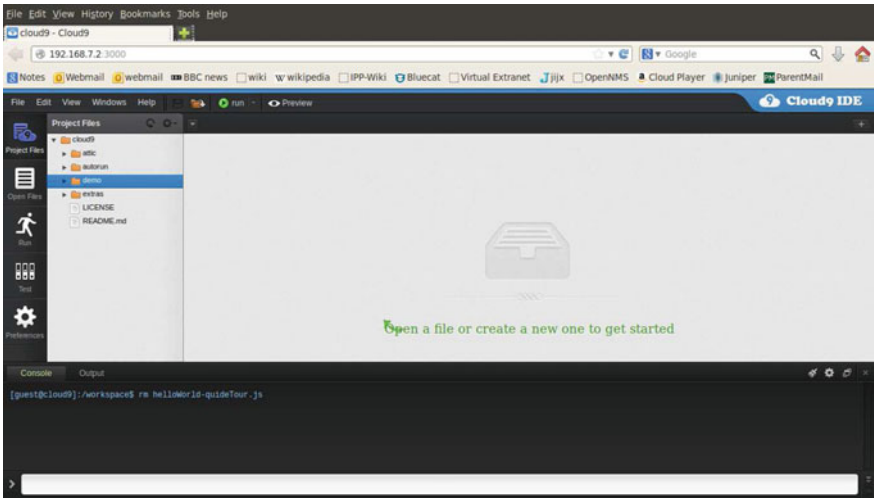


Fig. 7.14 Cloud9, IDE for the BeagleBone

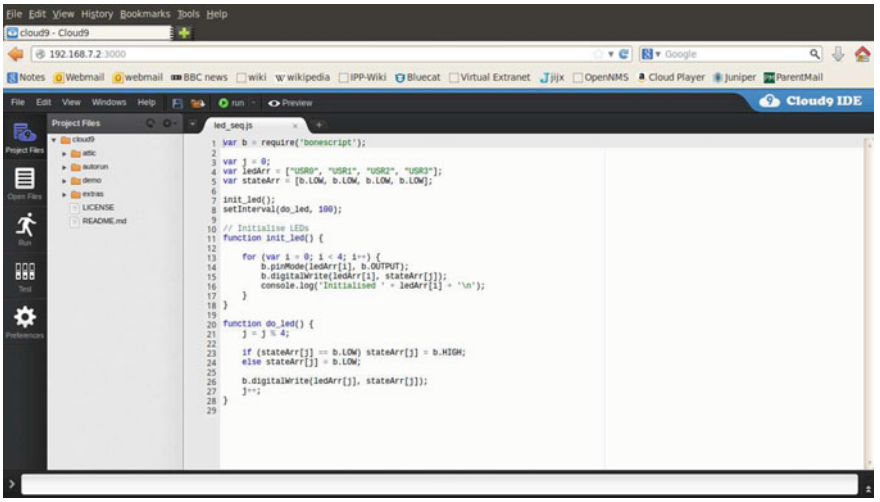


Fig. 7.15 Cloud9

Listing 7.3 *BeagleBone Black LED example*

```
var b = require('bonescript');

var j = 0;
var ledArr = ["USR0","USR1","USR2","USR3"];
var stateArr = [b.LOW, b.LOW, b.LOW, b.LOW];

init_led();
setInterval(do_led, 100);

// Initialise LEDs
function init_led() {

    for (var i = 0; i < 4; i++) {
        b.pinMode(ledArr[i], b.OUTPUT);
        b.digitalWrite(ledArr[i], stateArr[j]);
        console.log('Initialised ' + ledArr[i] + '\n');
    }

}

// Toggle LEDs
function do_led() {
    j = j % 4;
```

```
        if (stateArr[j] == b.LOW) stateArr[j] = b.HIGH;
        else stateArr[j] = b.LOW;

        b.digitalWrite(ledArr[j], stateArr[j]);
        j++;
    }
```

7.3 Summary

The Raspberry Pi and the BeagleBone are two ARM based computer systems. The Raspberry Pi was initially conceived as low-cost desktop PC, but is small footprint lends itself to embedded projects. Indeed, there are many examples published on the World-Wide Web of embedded projects using the Raspberry Pi (for more information look at social media sites such as Facebook and Twitter).

The BeagleBone is more suited to embedded projects. However, the BeagleBone Black appears to be aimed specifically at the Raspberry Pi market, as it can also operate as a personal computer.

References

1. Raspberry Pi Foundation. Raspberry Pi. http://www.raspberrypi.org/. Accessed 14 May 2013
2. Holt A (2013) Computer renaissance. http://www.ip-performance.co.uk/blog/raspberrypi1. Accessed 20 June 2013
3. Beagleboard. Beagleboard.org—community support open hardware. Accessed 14 May 2013
4. RISC OS Open. RISC OS Open. https://www.riscosopen.org/content. Accessed 21 June 2013
5. RISC OS Org. RISC OS Org. http://www.riscos.org.uk. Accessed 21 June 2013
6. Pike R, Presotto D, Thompson K, Trickey H (1990) Plan 9 from bell labs. In: Proceedings of the Summer 1990 UKUUG Conference, pp 1–9
7. Adafruit. Adafruit's Raspberry Pi Lesson 5. using a console cable. http://learn.adafruit.com/adafruits-raspberry-pi-lesson-5-using-a-console-cable/connect-the-lead. Accessed 25 June 2013
8. MODMYPI. Slice of Pi - Raspberry Pi breakout board. https://www.modmypi.com/slice-of-pi-raspberry-pi-breakout-boad. Accessed 25 June 2013
9. Cool Components. UartSBee V4—XBee Adapter. http://www.coolcomponents.co.uk/catalog/uartsbee-xbee-adapter-p-980.html. Accessed 2 July 2013

OpenWRT

8

OpenWRT is GNU/Linux distribution for embedded systems. However, it is not merely a static firmware image, it is a complete framework for building customised firmware images. The images comprise a bootloader, kernel, root filesystem and applications.

The OpenWRT project as means of developing third party firmware for The Linksys WRT54G devices. The Linksys WRT54G is a home router with built-in firewall and WiFi access-point. There are a number of variiants, namely the WRT54GL and WRTSL54GS. The significance of this product is that is used General Public License (GPL) code for its firmware. Given this, Linksys were "encouraged" to release this firmware back into the community (pursuant with the license agreement). This enabled the community to build their own firmware images for WRT54G devices. OpenWRT, as well as several other projects, was developed for the purpose of building WRT54G firmware images. OpenWRT, however, is not limited to just Linksys WRT54G routers, it supports a plethora of embedded devices, see [1] for a comprehensive list. At the time of writing this book, there are a number of branches OpenWRT which are listed in Table 8.1.

In this chapter we introduce two devices which run OpenWRT, Open-mesh and Dragino. Both units are based upon the MIPS processor (microprocessor without interlocked pipeline stages). The MIPS processor is a RISC architecture developed by MIPS Technologies. MIPS processors are used widely in embedded environments and is often used in educational environments for the purpose of studying microprocessors.

Open-mesh is sold as commercial mesh WiFi solution but the is it possible to re-flash devices with firmware for a customised embedded application. The Dragino is a similar device to the Open-mesh but more aimed at the Arduino Yùn market which is a device that combines a microcontroller with a WiFi system on a chip (SoC) that can run GNU/Linux.

A. Holt and C.-Y. Huang, *Embedded Operating Systems*, Undergraduate Topics in Computer Science, DOI 10.1007/978-1-4471-6603-0_8

Table 8.1 Branches of OpenWRT (at the time of writing this book)

Name	Verion	Kernel	Comment
White Russian	0.9	2.4	Old version
Kamikaze	7.06–8.09.2	2.6	Old verson
Backfire	10.03 and 10.03.1	2.6.32	Old version still supported
Attitude adjustment	12.09	3.3	Current version
Barrier breaker			Development

8.1 Open-Mesh

In this section, we demonstrate building OpenWRT for Open-Mesh OMxP devices [2]. The units are sold as WiFi mesh devices but can be re-flashed with custom OpenWRT firmware (see Fig. 8.1). Table 8.2 shows Open-Mesh OMxP range (at the time of writing this book).

Fig. 8.1 Open-mesh WiFi router

Table 8.2 Open-mesh models

Model	System Tyoe	CPU Model	Speed (MHz)	Comment
OM1P	Atheros AR2315	MIPS 4KEc V6.4	183	Legacy hardware
OM2P	Atheros AR7240	MIPS 24Kc V7.4	400	
OM2P-LC	Atheros AT9330	MIPS 74Kc V4.12	400	Low-cost
OM2P-HS	Atheros AR9341	MIPS 24Kc V7.4	520	High-speed

8.1.1 Building OpenWRT

OpenWRT can be downloaded and installed using the software version control system, Subversion. The easiest way to install Sunbversion is with your distributions package management utility. For example, on a Debian based system, Subversion can be installed with the command-line:

```
$ sudo apt-get install subversion
```

Similarly, install libncurses5-dev, ncurses-term, zlib1g-dev, gawk, bison, flex, autoconf, unzip and gettext.

The OpenWRT Buildroot comprises a set of Makefiles and patches for building a cross-compiler toolchain and a root filesystem for the target system. Download the OpenWRT Buildroot with:

```
$ svn checkout  svn://svn.openwrt.org/\openwrt/trunk \
> attidude_adjustment
```

Set the BUILDROOT environment variable accordingly and make it the current directory:

```
$ export BUILDROOT=$HOME/attitude_adjustment
$ cd $BUILDROOT
```

The OpenWRT Buildroot consists of a number of feeds. Each feed is a collection of packages. Install OpenWRT feeds by changing directory to the OpenWRT Buildroot directory and running the feeds update script:

```
$ ./scripts/feeds update
```

Run the command-line below to ensure all the packages are available for selection:

```
$ make package/symlinks
```

OpenWRT is highly customisable. Packages can be selected for inclusion through a menu driven configuation utilty. To configure OpenWRT run:

```
$ make menuconfig
```

The command above starts the configuration script as shown in Fig. 8.2. From the menu you can select the Target System and the packages you want to install. The OM2P chipset is Atheros AR9285. Select the Target System menu item and then select the chipset:

```
(X) (X) Atheros AR7xxx/AR9xxx
```

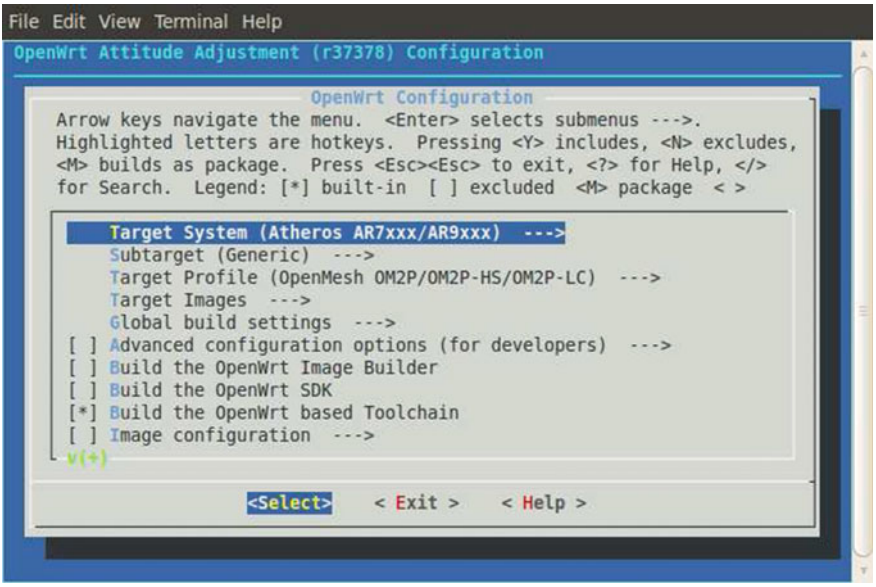


Fig. 8.2 OpenWRT make menu

If you have the legacy OM1P devices, which as Atheros AR2315 chipsets, then select the option Atheros AR231x/AR5312 Target System section.[1] If you do not have access to a physical system, you can build OpenWRT for User Mode Linux or a Xen vritual machine. For a Xen virtual machnine you should select "x86" for the Target System and then "Xen Paravirt Guest" from the Subtarget menu.

In this example, we use an OM2P-LC unit, therefore in the Target Profile we select the appropriate option:

```
(X) OpenMesh OM2P/OM2P-HS/OM2P-LC
```

The OM2P has watchdog hardware which will reboot the unit every five min if the timer is not reset. Fro this reason we select the `om-watchdog` option in the Base section:

```
<*> om-watchdog
```

[1] While this section focuses on Open-mesh devices, it is clear that OpenWRT supports a number of other systems. Discussing theses systems in detail is outside the scope of this book. However, the instructions in this section should be generic enough to use to compile firmware for other devices. One merely needs to pay attention to setting the Target System appropriately.

By default, the lightweight DNS/DHCP server `dnsmasq` is selected. As we do not want the unit to run either DNS or DHCP services, we unselect the option (also in the Base section):

```
< > dnsmasq
```

The default SSH (secure shell) utility is Dropbear. Dropbear, however, does not provide a secure FTP (SFTP) server. For this reason we install the OpenSSH server and SFTP server utilities. In the SSH subsection of the Network section, select `openssh-server` and: `openssh-sftp-server`

```
<*> openssh-server
<*> openssh-sftp-server
```

Finally, we select the option to build the cross toolchain for the MIPS architecture of the OM2P:

```
[*] Build the OpenWrt based Toolchain
```

Custom configuration files (such as those in */etc*) can be defined at the build stage. Create a directory called *files* in the backfire build directory:

```
$ mkdir files
```

Next, create some sub-directories in *files*:

```
$ mkdir files/{etc,etc/config}
```

The network interface configuration is specified in the host file directory *$BUILDROOT/files/etc/config/network* in Listing 8.1 (on the target directory, the file is */etc/config/network*).

Listing 8.1 *Contents of /etc/config/network*

```
config interface 'loopback'
option ifname 'lo'
option proto 'static'
option ipaddr '127.0.0.1'
option netmask '255.0.0.0'

config interface 'lan'
option ifname 'eth0'
option type 'bridge'
option proto 'static'
```

```
option ipaddr '172.16.50.236'
option gateway '172.16.50.1'
option netmask '255.255.255.0'
```

To compile the firmware image for the OM2P (and the cross toolchain), run the command below from the build directory [3]:

```
$ make
```

This will create the firmware image in the directory *bin/ar71xx*. You will find a number of files in this directory, but the one we are interested in is *openwrt-ar71xx-generic-om2p-squashfs-factory.bin*.

```
$ svn co http://dev.cloudtrax.com/downloads/svn/\
> ap51-flash/trunk/ ap51-flash

 pushd ap51-flash
~/ap51-flash ~

$ make
make -j 1 ap51-flash
make[1]: Entering directory '/home/aholt/ap51-flash'
    CC flash.o
    CC proto.o
    CC router_redboot.o
    CC router_tftp_client.o
    CC router_tftp_server.o
    CC router_types.o
    CC router_images.o
    CC socket.o
    CC commandline.o
    LD ap51-flash
strip ap51-flash
make[1]: Leaving directory '/home/aholt/ap51-flash'

$ popd
~
```

Connect the OM2P's Ethernet to your local switch but do *not* power on the OM2P at this point. Change to the *bin/atheros* sub-directory under the Backfire build directory and run `ap51-flash` to flash the firmware:

```
$ ~/ap51-flesh//ap51-flash eth0 \
> openwrt-ar71xx-generic-om2p-squashfs-factory.bin
```

Once the OM2P has booted up, we can Telnet to it and login as root without a password (at this stage):

```
$ telnet 169.254.105.57
root@OpenWrt:~/#
```

Set the root password:

```
root@OpenWrt:~/# passwd
```

Setting the password diables Telnet and enables SSH. Access to the device is now only through SSH. (though your current Telnet session will continue until you log out). The default SSH utilty is Dropbear. Later in this chapter we describe how to use the eclipse IDE to develop embedded applications on our OpenWRT system. For this we need the OpenSSH version of SSH. Here we describe how to change Dropbear for OpenSSH. We must first change the port that Dropbear listens on for SSH session:

```
root@OpenWrt:# uci set dropbear.@dropbear[0].Port=2022
root@OpenWrt:# uci commit dropbear
root@OpenWrt:# /etc/init.d/dropbear restart
```

We can now enable OpenSSH which will listen on SSH port 22:

```
root@OpenWrt:# /etc/init.d/sshd enable
root@OpenWrt:# /etc/init.d/sshd start
```

Remotely login using SSH, provide the root password when prompted:

```
$ ssh root@169.254.105.57
%root@OpenWrt:~# passwd aholt
%
```

8.2 Dragino

In this section we discuss the Dragino MS14 (see Fig. 8.3) which runs OpenWRT. In the previous chapter we briefly introduced the Arduino microcontroller. The Arduino is not controlled by an operating system like GNU/Linux. Arduinos are pre-programmed with a bootloader for uploading user programs to the on-chip flash memory. User programs are compiled into an executable cyclic executive program.

Fig. 8.3 Dragino

The Adruino Yùn is a ATmega32u4 microcontroller which also incorporates an IEEE 802.11n SoC (system on a chip) for WiFi and Ethernet communication. The AR9331 SoC is a MIPS processor and runs Linino which is a customised version of OpenWRT. The advanatge of this configuration is that the GNU/Linux can be used to upload skteches to the ATmega32u5 remotely. Furthermore, sketches running on the ATmega32u4 can communicate with the GNU/Linux system. This makes it possible for sketches to run shell scripts and use the WiFi and Ethernet interfaces.

Like the Arduino Yùn, the Dragino MP14 is based upon a AR9331 SoC. It runs the OpenWRT operating system, including the Arduino Yùn operating system, Linino. The Dragino does not come with a ATmega32u4 as standard. However, an ATmega32u processor is available as a plugin module, effectively, turning the Dragino MS14 into an Arduino Yùn. In this section we discuss the Dragino MS14 and the Arduino compatiblle plugin module, M32. There are two types of Dragino motherboard, the MS14-S and and MS14-P. The difference between the two is that, the MS14-S has a external terminal block so the GPIO ports can be accessed while motherboard is in the casing (see Fig. 8.4).

8.2.1 Booting up the Dragino for the First Time

The Dragino is shipped with an OpenWRT operating system pre-installed. Power up the Dragino and connect the Ethernet port labelled “LAN” to the Ethernet port of

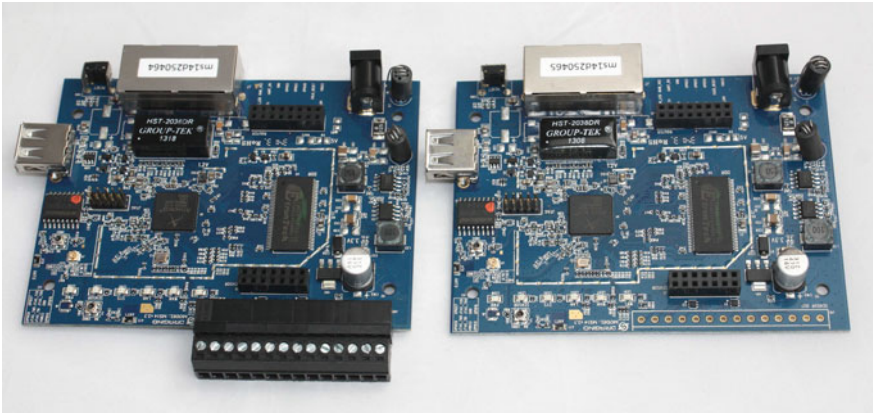


Fig. 8.4 The MS14-S (*left*) and MS14-P (*right*)

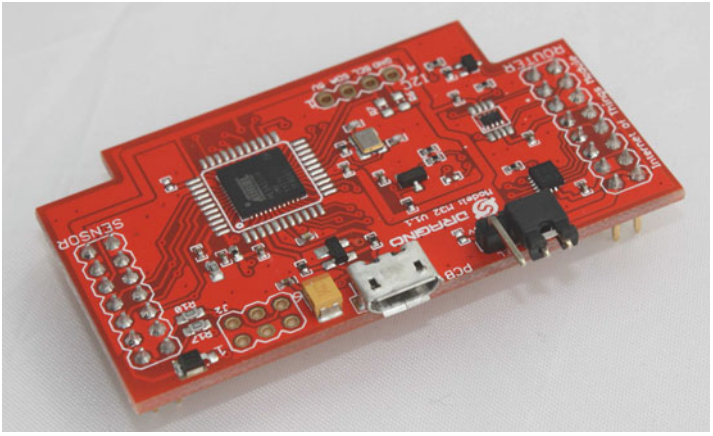


Fig. 8.5 MS32 module

a PC (see Fig. 8.5). The Dragino runs a DHCP service so the PC will bind to an IP address. If the PC does not receive any IP details automatically, issue the command:

```
$ sudo dhclient eth0
```

Run a web browser and enter http://10.130.1.1/ into the URL dialogue box. This will display the initial login page shown in Fig. 8.6. As this is the first time the unit has booted up, the root password is not set. To set a password, click the text "Go to password configuration..." on the initial login page. The web page shown in Fig. 8.7 is displayed.

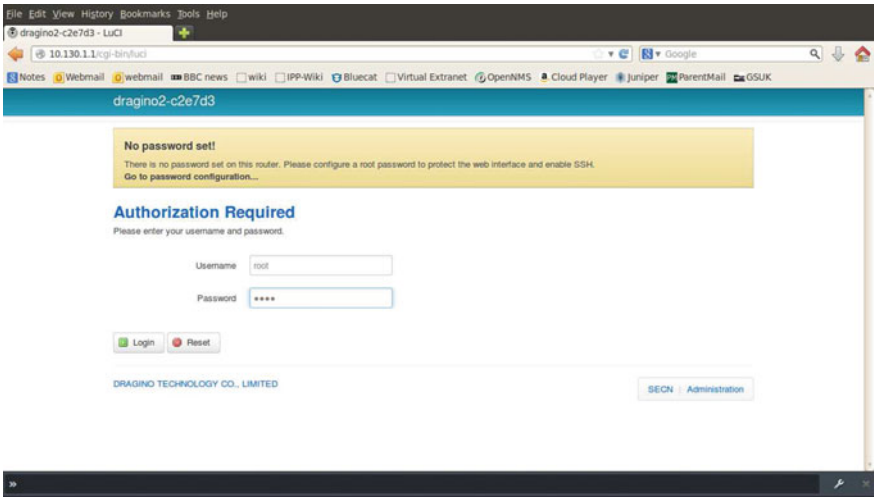


Fig. 8.6 Dragino login web page

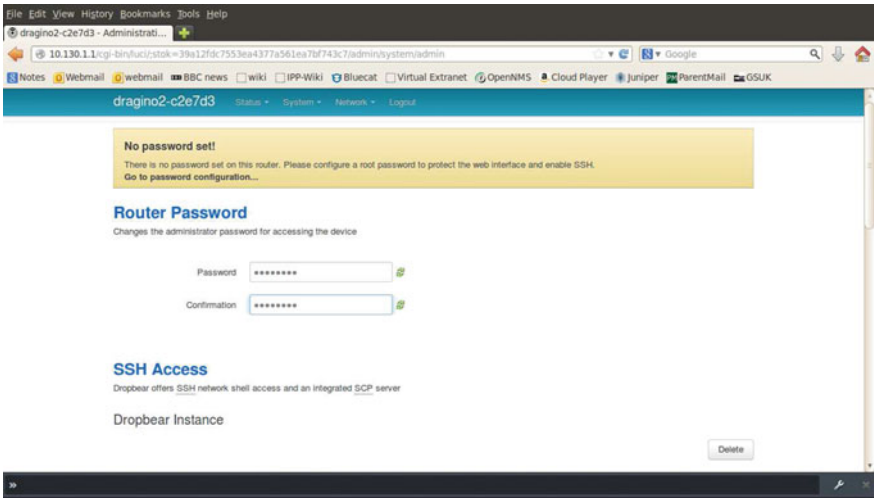


Fig. 8.7 Set the root password

Enter and password in the password field (and in the confirmation field). Scroll down to the bottom of the screen and click "Save and Apply". Setting a root password will also enable an SSH (secure shell) service on the unit. We can now access the command-line of the unit remotely (when prompted for a password, use the one you set above):

```
$ ssh root@10.130.1.1
root@10.130.1.1's password:


BusyBox v1.19.4 (2013-09-17 16:18:02 CST) built-in shell (ash)
Enter 'help' for a list of built-in commands.
  ___    ___    ___    ___  _ _  ___    ___
 |   \  |___\  |___|  | __    |  |   |  |   |
 |___/  |    |  |   |  |___|  _|_  |   |  |___|
-----------------------------------------------
W i F i, L i n u x, M C U, E m b e d d e d

OpenWRT ATTITUDE ADJUSTMENT (r33887)
Dragino-v2 MS14 1.0-IoT-1
Build Tue Sep 17 16:17:29 CST 2013

www.dragino.com
-----------------------------------------------------


root@dragino2-c2e7d3:~#
```

8.2.2 Running Arduino Yùn Firmware

In this subsection we show how to re-flash the Dragino with Linino version of OpenWRT which, enables the Dragino to operate as Arduino Yùn. Unlike the Arduino Yùn, the Dragino lacks a built in ATmega32u4 processor. Instead, Dragino provides an ATmega32u4 plugin module which can be programmed using the Arduino IDE. Table 8.3 shows the technical specification for the M32 module.

The firmware can be downloaded from the web site in [5]. Click on the link for the latest firmware (which is Beta1.3 at the time of writing this book). The kernel and root filesystem images are required:

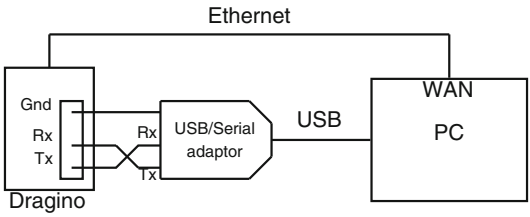
- ms14-arduino-yun-kernel-beta1.3.bin
- ms14-arduino-yun-rootfs-squashfs-beta1.3.bin

The firmware is installed using TFTP (trivial file transfer protolcol). Put the files above into */tftpboot* directory of TFTP server. The firmware is transferred to the Dragino by initiating a TFTP download from the Dragino's bootloader prompt. The bootloader prompt is accessible through the serial console. The Dragino's serial console can be accessed using the UART_TX and UART_RX pins on the J1 block. We used a USB to serial converter to connect a PC to the Dragino serial port (if

Table 8.3 M32 specification

Microcontroller	ATmega32u4
Operating voltage	5 V
I/O voltage	5 V
3.3 V output channel	1
5 V output channels	2
SRAM	2.5 KB
Flash memory	32 KB
EEPROM	1 KB
Clock speed	16 Mhz
Digital I/O pins	10
Analog input channels	8
PWM channels	4
ESD-protection	IEC 61000-4-2

Fig. 8.8 Arduino Yùn



you do not have a USB/serial converter then an Arduino can be adapted, refer to sect. 7.1.2). The TX pin (resp. RX pin) of the USB/Serial converter is connected to the UART_RX pin (resp. UART_TX pin) of the Dragino. We also connected the GND pins of the converter and Dragino. Figure 8.8 shows a diagram of the *crossover* console connection. We can now access the the serial console of the Dragino using a terminal emulator on the PC.

The diagram in Fig. 8.8 also shows the PC and Dragino connected via an Ethernet cable. We use the PC as a TFTP server from which the Dragino can download the firmware images.

Run the minicom terminal emulator:

```
$ sudo minicom /dev/ttyUSB0
```

The communication settings for the Dragino serial console are 1,15,200 baud, 8 bits with no parity, so the terminal emulator should be configured accordingly. Power cycle the Dragino. Boot procedure messages are displayed on the console:

```
U-Boot 1.1.4 (Aug 12 2012 - 20:14:51)
```

```
AP121-2MB (ar9330) U-boot
DRAM:  #### TAP VALUE 1 = f, 2 = 10
64 MB
Top of RAM usable for U-Boot at: 84000000
Reserving 212k for U-Boot at: 83fc8000
Reserving 192k for malloc() at: 83f98000
Reserving 44 Bytes for Board Info at: 83f97fd4
Reserving 36 Bytes for Global Data at: 83f97fb0
Reserving 128k for boot params() at: 83f77fb0
Stack Pointer at: 83f77f98
Now running in RAM - U-Boot at: 83fc8000
id read 0x100000ff
flash size 16777216, sector count = 256
Flash: 16 MB
In:    serial
Out:   serial
Err:   serial
Net:   ag7240_enet_initialize...
No valid address in Flash. Using fixed address
No valid address in Flash. Using fixed address
: cfg1 0x5 cfg2 0x7114
eth0: 00:03:7f:09:0b:ad
eth0 up
: cfg1 0xf cfg2 0x7214
eth1: 00:03:7f:09:0b:ad
athrs26_reg_init_lan
ATHRS26: resetting s26
ATHRS26: s26 reset done
eth1 up
eth0, eth1
```

Interrupt the boot process when the line below is displayed. This needs to be done with 4 s, otherwise, the boot process continues.

```
Hit any key to stop autoboot:  4
```

Interrupting the boot process gives you the bootloader prompt:

```
dr_boot>
```

Initially, the Dragino has a default IP address of 192.168.255.1 (with 255.255.255.0 subnet mask). It expects the TFTP server to have an address of 192.168.255.2. Set IP address of the TFTP server (which, in our case, is the PC in Fig. 8.8):

```
$ sudo ifconfig eth0 192.168.255.2 netmask \
> 255.255.255.0
```

From the boot prompt of the Dragino test the connectivity to the TFTP server:

```
dr_boot> ping 192.168.255.2
Using eth0 device
host 192.168.255.2 is alive
```

Upload the kernel image from the PC to the Dragino:

```
dr_boot> tftpboot 0x81000000 ms14-arduino-yun-kernel-bet
a1.3.bin
Using eth0 device
TFTP from server 192.168.255.2; our IP address is
192.168.255.1
Filename 'ms14-arduino-yun-kernel-beta1.3.bin'.
Load address: 0x81000000
Loading: ################################################
         ################################################
         ################################################
         ####################################
done
Bytes transferred = 1179648 (120000 hex)
```

Make a note of the bytes transferred (in this case 1,20,000 bytes in hexidecimal). Now erase a section of memory equal in size to the number of bytes transferred (note that the last parameter is the number of bytes in hexdecimal that was transferred when the kernel image was downloaded):

```
dr_boot> erase 0x9fea0000 +0x120000
Erase Flash from 0x9fea0000 to 0x9ffbffff in Bank # 1
First 0xea last 0xfb sector size 0x10000
 251
Erased 18 sectors
```

Copy the kernel to flash memory:

```
dr_boot> cp.b 0x81000000 0x9fea0000 $filesize
Copy to Flash... write addr: 9fea0000
done
```

Load the file containing the root filesystem (again making note of the number of bytes transferred):

```
dr-boot> tftpboot 0x81000000  ms14-arduino-yun-rootfs-s
quashfs-beta1.3.bin
Using eth0 device
TFTP from server 192.168.255.2; our IP address is
192.168.255.1
Filename 'ms14-arduino-yun-rootfs-squashfs-beta1.3.bin'.
Load address: 0x81000000
Loading: ################################################
         ################################################
         ################################################
         ################################################
         ################################################
         ################################################
         ################################################
         ################################################
         ################################################
         ################################################
         ################################################
         ################################################
         ################################################
         ################################################
         ################################################
         ################################################
         ################################
done
Bytes transferred = 5505024 (540000 hex)
```

Erase flash memory (last paramater is the byte count of the file transfer above):

```
dr_boot> erase 0x9f050000 +0x540000
Erase Flash from 0x9f050000 to 0x9f58ffff in Bank # 1
First 0x5 last 0x58 sector size 0x10000
  88
Erased 84 sectors
```

Copy to flash:

```
dr_boot> cp.b 0x81000000 0x9f050000 $filesize
Copy to Flash... write addr: 9f050000
done
```

Set the boot address:

```
dr_boot> setenv bootcmd bootm 0x9fea0000
```

Save enviroment:

```
dr_boot> saveenv
Saving Environment to Flash...
Protect off 9F030000 ... 9F03FFFF
Un-Protecting sectors 3..3 in bank 1
Un-Protected 1 sectors
Erasing Flash...Erase Flash from 0x9f030000 to
0x9f03ffff in Bank # 1
First 0x3 last 0x3 sector size 0x10000
   3
Erased 1 sectors
Writing to Flash... write addr: 9f030000
done
Protecting sectors 3..3 in bank 1
Protected 1 sectors
```

Reboot the unit:

```
dr_boot> reset
```

The Dragino will reboot but on this occaison do not interrupt the boot process. Once the operating system has booted, it will broadcast and open SSID with a prefix “Dragino-” followed by a sequence of hexdecimal digits. Our unit, for example, has the SSID Dragino-A84041134E3C. In order to test the unit, assoicate with the SSID then SSH to the root account of the unit (the default root password is “arduino”):

```
$ ssh root@192.168.240.1
root@192.168.240.1's password:

BusyBox v1.19.4 (2014-04-15 14:13:15 CST) built-in
shell (ash)
Enter 'help' for a list of built-in commands.


 _____  _____       _     ____ ___ _   _  ___
|  _ \|  _ \     / \   / ___|_ _| \ | |/ _ \
| | | | |_) |   / _ \| |  _ | ||  \| | | | |
| |_| |  _ <   / ___ \ |_| || || |\  | |_| |
|____/|_| \_\/_/   \_\____|___|_| \_|\___/

Firmware Version: Dragino-Yun-Beta1.3.1
Build Sat Apr 19 11:00:29 CST 2014

root@Arduino:~#
```

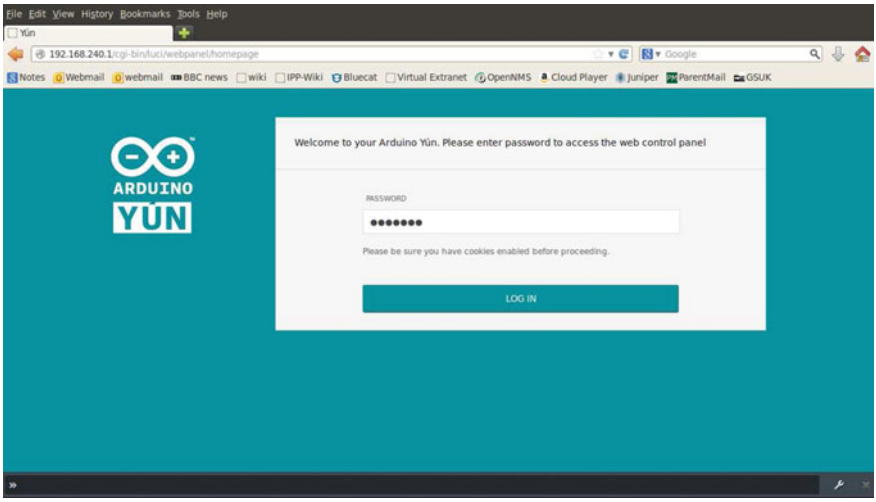


Fig. 8.9 Arduino login page

The Dragino can also be accessed using a web browser, enter http://192.168.240.1/ into URL dialogue box. Figure 8.9 shows the web login page. Enter "Arduino" into the password dialogue box and click the "LOG IN" button. This takes you to the Arduino Yùn welcome page (Fig. 8.10).

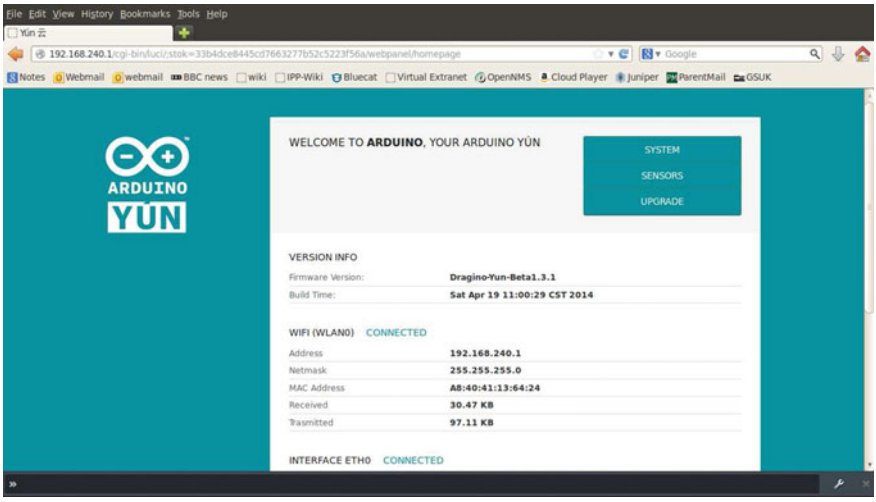


Fig. 8.10 Arduino Yùn welcome page

8.3 Programming the M32 Unit

The M32 unit is an Arduino compatible plugin unit for the Dragino. The M32 connects to the MS14 using the J1 and J5 expansion headers. Figure 8.11 shows the M32 modules plugged into the expansion headers of the Dragino Motherboard. Run the Arduino IDE and enter the sketch in Listing 8.2. This is a simple sketch that blinks an LED connected to Dragino's pin 2 (which is actually pin 4 on terminal block). (see Sect. 7.1 for instructions on how to install the Arduino IDE).

Listing 8.2 *Arduino Sketch for blinking an LED*

```
/*
  Flash led on pin 2
*/

int led = 2;

void setup() {
  pinMode(led, OUTPUT);
}

void loop() {
  digitalWrite(led, HIGH);
```

Fig. 8.11 The M32 module attached the Dragino motherboard

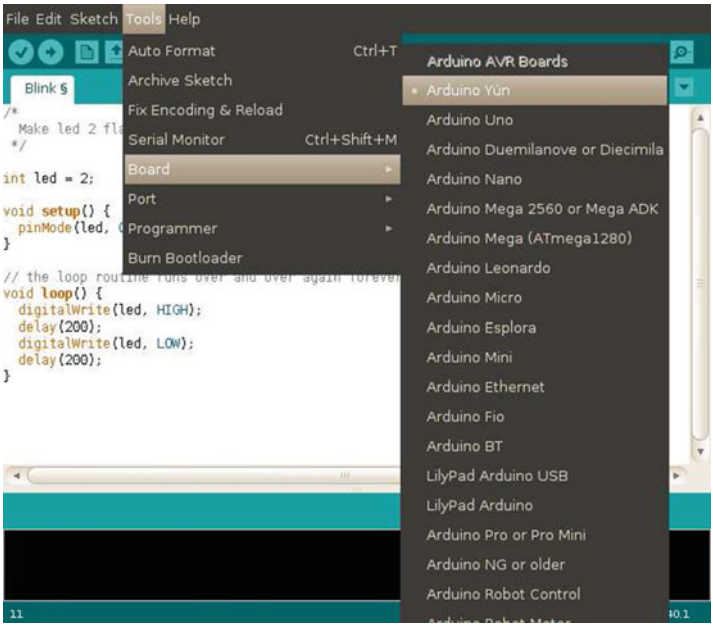


Fig. 8.12 Set the Arduino board type

```
    delay(200);
    digitalWrite(led, LOW);
    delay(200);
}
```

Before compiling the sketch, the Arduino board type and port needs to set. To set the board type, select "Tools" and then select "Board" from the drop down menu. Another drop down menu appears with a selection of Arduino board types. Select "Adruino Yùn" as show in Fig. 8.12. To set the port, select "Tools" and then "Port". From the drop down menu select "Arduino at 192.168.240.1 (Arduino Yùn)" as shown in Fig. 8.13.

The sketch can now be compiled, click on the button marked with a "✓". On sucessful compilation, download the compiled sketch to the M32 module by clicking on the "→" button. When prompted for a password, enter "Arduino" (Fig. 8.14 shows the password dialogue box).

Once the download of the (compiled) sketch to the M32 has completed, the sktech will run. To see the results of the sketch execution, connect an LED to the Dragino pins on the terminal block as shown in circuit diagram in Fig. 8.15. The LED will blink continuously three times a second.

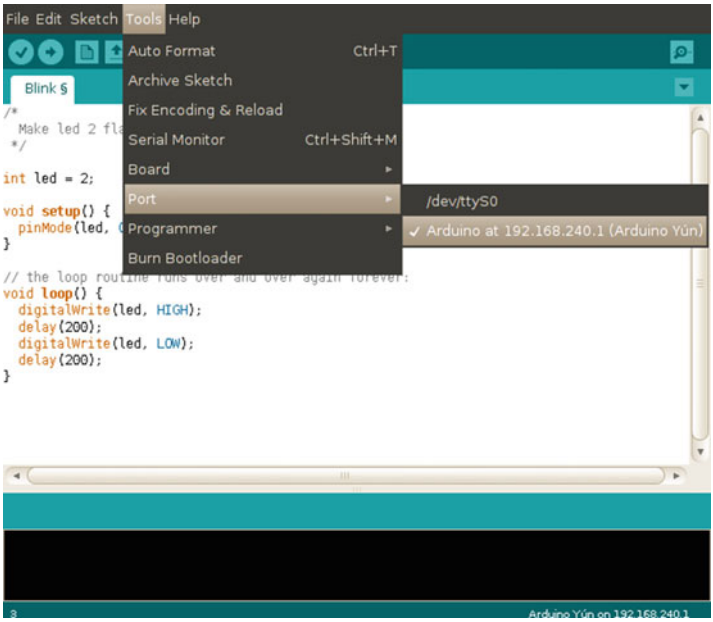


Fig. 8.13 Set the Arduino port

Fig. 8.14 Arduino password dialogue box

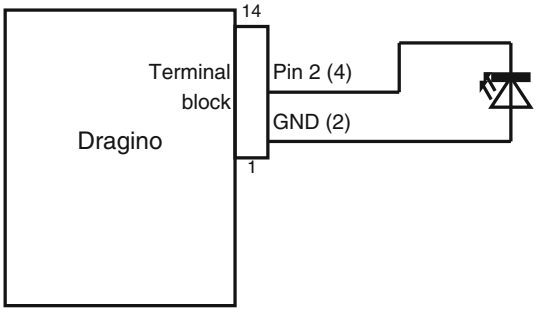


Fig. 8.15 LED circuit

8.4 Summary

OpenWRT is a complete framework for building GNU/Linux firmware images for embedded systems. OpenWRT is available for many platforms. In this chapter we looked two devices, namely, the Open-mesh OMxP devices and Dragino. The Open-mesh is sold as a WiFi mesh solution but its firmware can be reflashed with a custom OpenWRT operating system.

The Dragino can also be reflashed with custom OpenWRT firmware. The interesting faeture of the Dragino is that it can run the Linino which the custom firmware for the Arduino Yùn.

References

1. OpenWRT (2013) OpenWRT table of hardware. http://wiki.openwrt.org/toh/start. Accessed 21 June 2013
2. Open mesh UK (2013) Open mesh. http://www.openmesh-uk.com/. Accessed 4 Jan 2013
3. OpenWRT (2013) OpenWRT Network Configuration. http://wiki.openwrt.org/doc/uci/network. Accessed 4 Jan 2013
4. Mccoll James (2011) Breaking free the gumstix DSP. Linux J 6:52–56
5. Dragino (2014) Yun firmware download. http://www.dragino.com/downloads/index.php?dir=motherboards/ms14/Firmware/Yun/. Accessed 16 June 2014

Start-up Scripts

A

The start-up/shutdown scripts for Sect. 5.10 are listed in this Appendix. These files need to be created in the *$ROOT/etc/init.d* directory.

Listing A.1 *Contents of /etc/init.d/rcS*

```
#!/bin/sh

PATH=/sbin:/bin/:/usr/sbin:/usr/bin
runlevel=S
prevlevel=N
umask 022
export PATH runlevel prevlevel

# Read configuration variables
[ -f /etc/defaults/rcS ] && . /etc/defaults/rcS
export VERBOSE

trap ":" INT QUIT TSTP

for i in /etc/rcS.d/S??*
do
        # Ignore dangling symlinks
        [ ! -f "$i" ] && continue
        case "$i" in
                *.sh)
                        (
                                trap - INT QUIT TSTP
                                set start
                                . $i
                        )
                        ;;
                *)
```


A. Holt and C.-Y. Huang, *Embedded Operating Systems*, Undergraduate Topics in Computer Science, DOI 10.1007/978-1-4471-6603-0

```
                        $i start
                        ;;
        esac
done
```

Listing A.2 *Contents of* /etc/init.d/rc

```
#!/bin/sh

for i in /etc/rc${1}.d/K*; do
        [ ! -f $i ] && continue
        case "$i" in
                *.sh)
                        sh $i stop
                        ;;
                *)
                        $i stop
                        ;;
        esac
done

for i in /etc/rc${1}.d/S*; do
        [ ! -f $i ] && continue
        case "$i" in
                *.sh)
                        sh $i start
                        ;;
                *)
                        $i start
                        ;;
        esac
done
```

Listing A.3 *Contents of* /etc/init.d/mountall

```
#!/bin/sh

case $1 in
        start|"")
                /bin/mount -na
                /bin/mount -n -o remount,rw /
                ;;
        stop)
                /bin/umount -na
```

```
                ;;
        *)      echo "Usage: mountall [start|stop]" >&2
                exit 3
                ;;
esac
```

Listing A.4 shows the bootmisc script.

Listing A.4 *Contents of* /etc/init.d/bootmisc

```
#!/bin/sh

start_up () {

        # Create /var directory
        tar xf /var.tar

        # Save dmesg output
        /bin/dmesg >/var/log/dmesg

        # Create FIFO
        [ ! -p /dev/initctl ] && /usr/bin/mkfifo /dev/initctl

        # Create /var/run/utmp
        touch /var/run/utmp

}

case "$1" in
  start|"")
        start_up
        ;;
  stop)
        # No operation
        ;;
  *)
        echo "Usage: bootmisc [start|stop]" >&2
        exit 3
        ;;
esac
```

Listing A.5 *Contents of* /etc/init.d/hostname

```
#!/bin/sh

if [ -f /etc/hostname ]
then
       /bin/hostname -F /etc/hostname
fi
```

Listing A.6 *Contents of* /etc/init.d/network

```
#!/bin/sh

case "$1" in
        start)
                echo -n "Bring up network interfaces: "
                /sbin/ifup -a
                echo "done."
                ;;
        stop)
                echo -n "Take down network interfaces: "
                /sbin/ifdown -a
                echo "done."
                ;;
        force-reload|restart)
                echo -n "Reconfigure network interfaces: "
                ifdown -a
                ifup -a
                echo "done."
                ;;
esac
```

Listing A.7 *Contents of* /etc/init.d/syslogd

```
#!/bin/sh

UNAME=syslogd
UTIL=/sbin/${UNAME}
DESC=${UNAME}

if [ ! -x ${UTIL} ]; then
       echo Skipping ${DESC}.
       exit 2
fi
set -e
```

```
case "$1" in
       start)
               echo -n "Starting ${UNAME}: "
               ${UTIL}
               EXIT=$?
               if [ $EXIT == 0 ]; then
                       echo "${UNAME} started sucessfully."
               else
                       echo "${UNAME} failed."
               fi
               ;;
       stop)
               echo -n "Stopping ${UNAME}: "
               if killall ${UNAME}
               then
                       echo "${UNAME} stopped sucessfully."
               else
                       echo "${UNAME} failed."
               fi
               ;;
       *)
               echo "usage: $0 {start|stop}"
               exit 1;
esac

exit 0
```

Listing A.8 *Contents of* /etc/init.d/klogd

```
#!/bin/sh

UNAME=klogd UTIL=/sbin/${UNAME} DESC=${UNAME}

if [ ! -x ${UTIL} ]; then
       echo Skipping ${DESC}.
       exit 2
fi

set -e

case "$1" in
       start)
               echo -n "Starting ${UNAME}: "
```

```
                ${UTIL}
                EXIT=$?
                if [ $EXIT == 0 ]; then
                        echo "${UNAME} started sucessfully."
                else
                        echo "${UNAME} failed."
                fi
                ;;
        stop)
                echo -n "Stopping ${UNAME}: "
                if killall ${UNAME}
                then
                        echo "${UNAME} stopped sucessfully."
                else
                        echo "${UNAME} failed."
                fi
                ;;
        *)
                echo "usage: $0 {start|stop}"
                exit 1;
esac

exit 0
```

Listing A.9 *Contents of* /etc/init.d/telnet

```
#!/bin/sh

UNAME=telnetd
UTIL=/usr/sbin/${UNAME}
DESC="Telnet daemon"

if [ ! -x ${UTIL} ]; then
        echo Skipping ${DESC}.
        exit 2
fi

set -e

case "$1" in
        start)
                echo -n "Starting ${UNAME}: "
                ${UTIL}
                EXIT=$?
                if [ $EXIT == 0 ]; then
```

```
                        echo "${UNAME} started sucessfully."
                else
                        echo "${UNAME} failed."
                fi
                ;;
        stop)
                echo -n "Stopping ${UNAME}: "
                if killall ${UNAME}
                then
                        echo "${UNAME} stopped sucessfully."
                else
                        echo "${UNAME} failed."
                fi
                ;;
        *)
                echo "usage: $0 {start|stop}"
                exit 1;
esac

exit 0
```

Listing A.10 *Contents of* /etc/init.d/ntp

```
#!/bin/sh

UNAME=ntpd
UTIL=/usr/sbin/${UNAME}
DESC="NTP daemon"
RTC=/sbin/hwclock

SERVER=0.debian.pool.ntp.org

if [ ! -x ${UTIL} ]; then
        echo Skipping ${DESC}.
        ${RTC} -s   # set system clock from RTC
        exit 2
fi

set -e

case "$1" in
        start)
                echo -n "Starting ${UNAME}: "
                ${UTIL} -p ${SERVER}
                EXIT=$?
```

```
                if [ $EXIT == 0 ]; then
                        echo "${UNAME} started sucessfully."
                else
                        echo "${UNAME} failed."
                fi
                ;;
        stop)
                echo -n "Stopping ${UNAME}: "
                $RTC -w    # sync RTC with system clock
                if killall ${UNAME}
                then
                        echo "${UNAME} stopped sucessfully."
                else
                        echo "${UNAME} failed."
                fi
                ;;
        *)
                echo "usage: $0 {start|stop}"
                exit 1;
esac

exit 0
```

Listing A.11 *Contents of* /etc/init.d/halt

```
#!/bin/sh

echo -n "Halting... "
/sbin/halt -d2  -f
```

Listing A.12 *Contents of* /etc/init.d/reboot

```
#!/bin/sh

echo -n "Rebooting... "
/sbin/reboot -d2 -f
```

Inittab

B

This Appendix lists the contents of the */etc/inittab* file.

Listing B.1 *Contents of /etc/inittab*

```
# $Id: inittab,v 1.91 2002/01/25 13:35:21 miquels Exp $

id:2:initdefault:

si::sysinit:/etc/init.d/rcS

~:S:wait:/sbin/sulogin

l0:0:wait:/etc/init.d/rc 0
l1:1:wait:/etc/init.d/rc 1
l2:2:wait:/etc/init.d/rc 2
l3:3:wait:/etc/init.d/rc 3
l4:4:wait:/etc/init.d/rc 4
l5:5:wait:/etc/init.d/rc 5
l6:6:wait:/etc/init.d/rc 6
z6:6:respawn:/sbin/sulogin

ca:12345:ctrlaltdel:/sbin/shutdown -t1 -a -r now

pf::powerwait:/etc/init.d/powerfail start
pn::powerfailnow:/etc/init.d/powerfail now
po::powerokwait:/etc/init.d/powerfail stop

1:2345:respawn:/sbin/getty 38400 tty1
2:23:respawn:/sbin/getty 38400
tty2 3:23:respawn:/sbin/getty 38400 tty3
```


A. Holt and C.-Y. Huang, *Embedded Operating Systems*, Undergraduate Topics in Computer Science, DOI 10.1007/978-1-4471-6603-0

```
4:23:respawn:/sbin/getty 38400 tty4
5:23:respawn:/sbin/getty 38400 tty5
6:23:respawn:/sbin/getty 38400 tty6

T0:23:respawn:/sbin/getty -L ttyS0 19200 vt100
T1:23:respawn:/sbin/getty -L ttyS1 19200 vt100
```

Index

A
Advanced RISC machine (ARM), 137
Android, 7
Angstrom, 7
Apple, 5
  Mac OS X, 5
  OS X, 116
Arch Linux, 7, 139
Arduino, 140, 167
ARM, *see* Advanced RISC machine
ARM instruction simulator (ARMulator), 134
ARMulator, *see* ARM instruction simulator
Artheros, 164
Assembler, 115, 121
AT&T
  Bell Laboratories, 5
ATmega32u4, 167

B
Bash, *see* Bourne again shell
BeagleBone, 151–160
Berkeley FFS, *see* Fast filesystem
Berkeley System Distribution, 5
Block devices, 44
BlueRay, 138
Bochs, 8
Boot block, 55
Boot sector, 60
Bootloader, 55
Bourne again shell (Bash), 88
BSD, 5, 116
  FreeBSD, 5
  NetBSD, 5
  OpenBSD, 5
Busybox, 88, 93
Bytecode, 115

C
C programming language, 5, 116
C++, 116
Character devices, 44
CHS, *see* Cylinder-head-sector
Code generator, 115
Commands
  ap51-flash, 166
  apt-get, 74, 89, 110
  as, 122, 130
  brctl, 85
  brtcl, 84
  chgrp, 84
  chmod, 43, 84
  chroot, 76, 104
  cpp, 118
  dd, 80, 104
  debootstrap, 73–75
  dpkg-reconfigure, 77
  fdisk, 62, 107
  ifconfig, 84
  iptables, 86, 155
  kill, 21
  ldd, 128
  losetup, 64, 65
  ls, 130
  make, 110
  mk2efs, 105
  mkdosfs, 65
  mkfs.ext2, 66, 80, 109
  mknod, 97
  mount, 65, 66, 80, 84, 109
  passwd, 77, 104
  pwd, 77
  screen, 82, 150
  stat, 44, 65, 66
  strip, 92, 93

A. Holt and C.-Y. Huang, *Embedded Operating Systems*, Undergraduate Topics in Computer Science, DOI 10.1007/978-1-4471-6603-0

tar, 80
Telnet, 167
telnet, 113
touch, 43
tunctl, 85
umount, 84, 107
wget, 94
who, 17
Compiler, 115
assembler, 121
ccp, 117
linker, 122
Cylinder-head-sector (CHS), 59

D
Debian, 73
Demountable volumes, 40
DHCP, *see* Dynamic host control protocol
Directories, 39
Disk partitions, 39
Dnsmasq, 165
Dragino, 167–179
M32, 168
MS14, 168
Dynamic host control protocol (DHCP), 83

E
ELF, *see* Executable and linkable format
Executable and linkable format (ELF), 74
EXTLINUX, 15, 110

F
Fast filesystem (FFS), 54
FFS, *see* Fast filesystem
FHS, *see* Filesystem Hierarchy Standard
FIFO, *see* First in first out
File permissions, 42
Files
/etc/config/network, 166
/etc/env, 101
/etc/fstab, 99
/etc/group, 98
/etc/inittab, 192
/etc/ld.so.conf, 92
/etc/nsswitch.conf, 99
/etc/passwd, 98
/etc/profile, 101
/etc/services, 99
extlinux.conf, 111
fstab, 75
hostname, 76
inittab, 76
interfaces, 76, 84, 99
Filesystem, 39
blocks, 54
boot block, 55
inodes, 55
superblock, 55
Filesystem Hierarchy Standard (FHS), 19
First in first out (FIFO), 39
Fortran, 116
FreeBSD, 5

G
GCC, *see* GNU compiler collection
GDB, *see* GNU debugger
General purpose input/output (GPIO), 137, 140
Glibc, 87
Glibc , *see* GNU C library
GNU C library (Glibc), 89
GNU Compiler Collection (GCC), 116
GNU debugger (GDB), 134, 135
GPIO, *see* General purpose input/output
Grand unified bootloader (GRUB), 15
GRUB, *see* Grand unified bootloader

H
Here documents, 26

I
IEEE 802.15.4, 144
Inodes, 55
Internet-of-things (IoT), 1
IoT, *see* Internet-of-things
Iptables, 155
ISOLINUX, 15

J
Java, 116
JavaScript, 160
Jobs, Steve, 5

K
Kernel
compilation, 78, 106
modules, 79

L
LBA, *see* Logical block addressing
Libraries
libncurses.so.5.4, 93

Lilo, *see* Linux loader
Linino, 168
Linker, 115, 122
Linux loader (LILO), 15
Locales, 77
Logical block addressing (LBA), 59

M
Mac OS X, 116
Master boot record (MBR), 14, 60
McIlory, Douglas, 5

N
Named pipes, *see* First in first out
NAT, *see* Network address translation
Ncurses, 87, 92
NetBSD, 5
Netfilter, 86
Network address translation (NAT), 85, 155
Network time protocol (NTP), 103, 143
Newlib, 132
Nextstep, 5
NTP, *see* Network time protocol

O
OM2P Watchdog timer, 164
OpenBSD, 5
OpenWRT, 7, 161–181

P
Packet forwarding, 155
Pascal, 116
Plan 9, 139
Preboot execution environment (PXE), 15
Preprocessor, 115, 117
Procfs, 58
Psuedo filesystems, 57
PXE, *see* Preboot execution environment
PXELINUX, 15

Q
Qemu, *see* Quick emulator
Quick emulator (Qemu), 8

R
Rapberian, 139
Raspberry Pi, 151
Real-time operating system (RTOS), 4
Reduced instruction set computer (RISC), 137
Rirchie, Dennis, 5
RISC, *see* Reduced instruction set computer
Risc OS, 139
RTLinux, 4
RTOS, *see* Real-time operating system
Runlevels, 16

S
SBC, *see* Single board computer
Scripts
  default.script, 101
  uml.sh, 81
Single board computer (SBC), 4
Slice of Pi, 140
SoC, *see* System on a chip
Sockets, *see* Unix sockets
Soekris, 88
Soft links, *see* Symbolic links
Start-up scripts
  /etc/init.d/bootmisc, 185
  /etc/init.d/halt, 190
  /etc/init.d/hostname, 186
  /etc/init.d/klogd, 188
  /etc/init.d/mountall, 185
  /etc/init.d/network, 186
  /etc/init.d/ntp, 190
  /etc/init.d/reboot, 190
  /etc/init.d/syslogd, 187
  /etc/init.d/telnet, 189
Subversion, 163
Sun Microsystems, 5
  Solaris, 5
  SunOS, 5
Superblock, 55
Symbolic links, 39, 52
Sysfs, 58, 156
SYSLINUX, 15, 110
System on a chip (SoC), 4, 168
Sysvinit, 16, 88

T
Thompson, Ken, 5
Trivial file transfer protocol (TFTP), 172
TUN/TAP, 84

U
UFS, *see* Unix filesystem
UML, *see* User mode Linux
University of California, 5
Unix, 5
  System III, 5
  System V, 5

System V Release 4, 5
Unix filesystem (UFS), 54
Unix sockets, 39
User mode Linux (UML), 8, 74, 78, 104

V
VFS, *see* Virtual filesystem
Virtual filesystem (VFS), 53
VMware, 8
Volume boot record (VBR), 14, 60

W
Wine, *see* Wine is not an emulator
Wine is not an emulator (Wine), 145

X
X-CTU, 145
XBee modules, 144
XEN, 8
Xenix, 5

Z
ZigBee, 144