Embedded Programming with Android™
The Android Deep Dive Series is for intermediate and expert developers who use Android Studio and Java, but do not have comprehensive knowledge of Android system-level programming or deep knowledge of Android APIs. Readers of this series want to bolster their knowledge of fundamentally important topics.

Each book in the series stands alone and provides expertise, idioms, frameworks, and engineering approaches. They provide in-depth information, correct patterns and idioms, and ways of avoiding bugs and other problems. The books also take advantage of new Android releases, and avoid deprecated parts of the APIs.

About the Series Editor

Zigurd Mednieks is a consultant to leading OEMs, enterprises, and entrepreneurial ventures creating Android-based systems and software. Previously he was chief architect at D2 Technologies, a voice-over-IP (VoIP) technology provider, and a founder of OpenMobile, an Android-compatibility technology company. At D2 he led engineering and product definition work for products that blended communication and social media in purpose-built embedded systems and on the Android platform. He is lead author of *Programming Android* and *Enterprise Android*. 
Embedded Programming with Android™

Bringing Up an Android System from Scratch

Roger Ye

Addison-Wesley

New York • Boston • Indianapolis • San Francisco
Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City
To the programmers who have great interest in embedded systems and the latest computing devices.
This page intentionally left blank
Contents in Brief

Preface xv
Acknowledgments xxi
About the Author xxiii

I Bare Metal Programming 1
1 Introduction to Embedded System Programming 3
2 Inside Android Emulator 13
3 Setting Up the Development Environment 25
4 Linker Script and Memory Map 39
5 Using the C Language 63
6 Using the C Library 93
7 Exception Handling and Timer 125
8 NAND Flash Support in Goldfish 183

II U-Boot 217
9 U-Boot Porting 219
10 Using U-Boot to Boot the Goldfish Kernel 249

III Android System Integration 281
11 Building Your Own AOSP and CyanogenMod 283
12 Customizing Android and Creating Your Own Android ROM 309

IV Appendixes 339
A Building the Source Code for This Book 341
B Using Repo in This Book 355
Index 359
This page intentionally left blank
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>xv</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>xxi</td>
</tr>
<tr>
<td>About the Author</td>
<td>xxiii</td>
</tr>
</tbody>
</table>

## I Bare Metal Programming  1

1. **Introduction to Embedded System Programming**  3  
   What Is an Embedded System?  3  
   Bare Metal Programming  3  
   Learning Embedded System Programming  5  
   Software Layers in an Embedded System  7  
   Tools and Hardware Platform  11  
   The Difference between Virtual Hardware and Real Hardware  11  
   Summary  12  

2. **Inside Android Emulator**  13  
   Overview of the Virtual Hardware  13  
   Configuring Android Virtual Devices  14  
   Hardware Interfaces  17  
   Serial  18  
   Timer  18  
   Summary  24  

3. **Setting Up the Development Environment**  25  
   The Host and Client Environments  25  
   Development Environment Setup  26  
   Downloading and Installing Android SDK  27  
   Downloading and Installing the GNU Toolchain for ARM  27  
   Integrated Development Environment  29  
   Your First ARM Program  29  
   Building the Binary  30  
   Running in the Android Emulator  32  
   makefile for the Example Projects  36  
   Summary  38  

4 Linker Script and Memory Map 39
   Memory Map 39
   Linker 41
      Symbol Resolution 42
      Relocation 46
      Section Merging 49
      Section Placement 50
   Linker Script 51
      Linker Script Example 53
   Initializing Data in RAM 56
      Specifying Load Address 58
      Copying .data to RAM 58
   Summary 61

5 Using the C Language 63
   C Startup in a Bare Metal Environment 63
      Stack 65
      Global Variables 68
      Read-Only Data 68
      Startup Code 68
   Calling Convention 78
      Calling C Functions from Assembly Language Code 79
      Calling Assembly Language Functions from C Code 81
   Goldfish Serial Port Support 81
      Check Data Buffer 87
      Data Input and Output 88
      Unit Test of Serial Functions 90
   Summary 92

6 Using the C Library 93
   C Library Variants 93
      C Library Variants in an Operating System 93
      C Library Variants in Bare Metal Environment 94
   Newlib C Library 96
   Common Startup Code Sequence 97
   CS3 Linker Scripts 97
   Customized CS3 Startup Code for the Goldfish Platform 103
System Call Implementations 104
Running and Debugging the Library 112
Using Newlib with QEMU ARM Semihosting 116
  Semihosting Support in Newlib C 117
  Semihosting Example Code 118
Summary 122

7 Exception Handling and Timer 125
  Goldfish Interrupt Controller 125
  The Simplest Interrupt Handler 128
    Interrupt Support Functions 129
      Implementation of the Simplest Interrupt Handler 132
  Nested Interrupt Handler 140
    Implementation of the Nested Interrupt Handler 142
    Testing Nested Interrupts and Discovering the Processor Mode Switch 155
  Testing System Calls/Software Interrupts 163
  Timer 164
    Goldfish-Specific Timer Functions 172
    U-Boot API 172
  Real-Time Clock 172
    Unit Test of Timer and RTC 174
Summary 181

8 NAND Flash Support in Goldfish 183
  Android File System 183
  NAND Flash Properties 185
  NAND Flash Programming Interface in the Goldfish Platform 187
  Memory Technology Device Support 188
  MTD API 189
    U-Boot API to Support NAND Flash 205
    Goldfish NAND Flash Driver Functions 205
  NAND Flash Programming Interface Test Program 206
    NAND Flash Information from the Linux Kernel 206
    NAND Flash Test Program 210
Summary 216
II U-Boot 217

9 U-Boot Porting 219
  Introducing U-Boot 219
  Downloading and Compiling U-Boot 220
  Debugging U-Boot with GDB 224
  Porting U-Boot to the Goldfish Platform 227
    Creating a New Board 228
    Processor-Specific Changes 229
    Board-Specific Changes 229
    Device Driver Changes 239
  Summary 246

10 Using U-Boot to Boot the Goldfish Kernel 249
  Building the Goldfish Kernel 249
  Prebuilt Toolchain and Kernel Source Code 250
  Running and Debugging the Kernel in the Emulator 252
  Booting Android from NOR Flash 254
    Creating the RAMDISK Image 256
    Creating the Flash Image 258
    Booting Up the Flash Image 258
    Source-Level Debugging of the Flash Image 266
  Booting Android from NAND Flash 270
    Preparing system.img 270
    Booting from NAND Flash 271
  Summary 280

III Android System Integration 281

11 Building Your Own AOSP and CyanogenMod 283
  Introducing AOSP and CyanogenMod 283
  Setting Up an Android Virtual Device 284
  AOSP Android Emulator Build 288
    AOSP Build Environment 288
    Downloading the AOSP Source 289
    Building AOSP Android Emulator Images 290
    Testing AOSP Images 292
  CyanogenMod Android Emulator Build 297
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downloading the CyanogenMod Source</td>
<td>297</td>
</tr>
<tr>
<td>Building CyanogenMod Android Emulator Images</td>
<td>298</td>
</tr>
<tr>
<td>Testing CyanogenMod Images</td>
<td>302</td>
</tr>
<tr>
<td>Summary</td>
<td>307</td>
</tr>
</tbody>
</table>

12 Customizing Android and Creating Your Own Android ROM 309

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supporting New Hardware in AOSP</td>
<td>309</td>
</tr>
<tr>
<td>Building the Kernel with AOSP</td>
<td>317</td>
</tr>
<tr>
<td>Building U-Boot with AOSP</td>
<td>322</td>
</tr>
<tr>
<td>Booting Android with U-Boot from NAND Flash</td>
<td>323</td>
</tr>
<tr>
<td>Supporting New Hardware in CyanogenMod</td>
<td>332</td>
</tr>
<tr>
<td>Building the Kernel with CyanogenMod</td>
<td>334</td>
</tr>
<tr>
<td>Building U-Boot and Booting Up CyanogenMod</td>
<td>337</td>
</tr>
<tr>
<td>Summary</td>
<td>338</td>
</tr>
</tbody>
</table>

IV Appendixes 339

A Building the Source Code for This Book 341

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setting Up the Build Environment</td>
<td>341</td>
</tr>
<tr>
<td>Setting Up a Virtual Machine</td>
<td>344</td>
</tr>
<tr>
<td>Organization of Source Code</td>
<td>344</td>
</tr>
<tr>
<td>Source Code for Part I</td>
<td>345</td>
</tr>
<tr>
<td>Building and Testing from the Command Line</td>
<td>345</td>
</tr>
<tr>
<td>Building and Testing in Eclipse</td>
<td>346</td>
</tr>
<tr>
<td>Source Code for Part II</td>
<td>350</td>
</tr>
<tr>
<td>Source Code for Part III</td>
<td>352</td>
</tr>
<tr>
<td>Building AOSP</td>
<td>352</td>
</tr>
<tr>
<td>Building CyanogenMod</td>
<td>353</td>
</tr>
</tbody>
</table>

B Using Repo in This Book 355

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resources for Repo</td>
<td>355</td>
</tr>
<tr>
<td>Syncing a New Source Tree In Minutes</td>
<td>355</td>
</tr>
<tr>
<td>Downloading Git Repositories Using Local Manifest</td>
<td>356</td>
</tr>
</tbody>
</table>

Index 359
This page intentionally left blank
Preface

Computing is becoming more and more pervasive. Computing devices are evolving from traditional desktop computers to tablets and mobile devices. With the newer platforms, embedded computing is playing a more important role than the traditional mainframe- and desktop-based computing. Embedded system programming looks very different in various usage scenarios. In some cases, it consists of application programming using the assembly and C languages on top of the hardware directly. In other cases, it takes place on top of a real-time operating system (RTOS). In the most complicated case, it can be a desktop-based system using a modern operating system such as Linux or Windows.

Due to the many different usage scenarios and hardware architectures that are possible, it is very difficult to teach embedded programming in a standard way in a school or university. There are simply too many hardware platforms based on a multitude of very different architectures. The processors or microprocessors can be as simple as 8-bit models or as complicated as 32-bit or even 64-bit devices. In most cases, students learn about embedded programming on a dedicated hardware reference board and use the compiler and debugger from a particular company. Obviously, this kind of development environment is unique and difficult to duplicate. To overcome these challenges, this book uses virtualization technology and open source tools to provide a development environment that any programmer can easily obtain from the Internet.

Who Should Read This Book

If you want to learn embedded system programming, especially embedded system programming on Android, this is the book for you. For starters, you may want to get some hands-on experience while you read a book. This book includes plenty of examples for you to try out. The good thing is that you don’t need to worry about having a hardware platform or development tools. All examples in this text are built using open source tools that you can download from the Internet, and all of them can be tested on the Android emulator. The source code is hosted in GitHub. Appendix A describes the build environment setup and explains how to work with the source code in GitHub.

Note

Git is a version control tool used by many open source projects. If you are new to it, you can search for “git” or “GitHub” on the Internet to find tutorials on its use. A free book on GitHub, *Pro Git* by Scott Chacon, can also be downloaded from the following address:


GitHub is a free git repository on the Internet that can be used to host open source projects. You can find the git repositories in this book at the following address:

https://github.com/shugaoye/
If you have just started your career as an embedded system software engineer, your first project may be porting U-Boot to a new hardware platform. This book gives you the detailed steps on how to port U-Boot to the Android emulator.

If you are an experienced software developer, you may know that it is quite difficult to debug a complex device driver in your project. In this book, we explore a way to separate the debugging of the hardware interface from the device driver development. We explain how to debug serial ports, interrupt controllers, timers, the real-time clock, and NAND flash in a bare metal environment. We then explain how to integrate these examples with U-Boot drivers. The same method can also be used for Linux or Windows driver development.

To take full advantage of this book, you should be familiar with the C language, basic operating system concepts, and ARM assembly language. Ideally, readers will be graduates in computer science or experienced software developers who want to explore low-level programming knowledge. For professionals who work on Android system development, this is also a good reference book.

How This Book Is Organized

In this book, we discuss the full spectrum of embedded system programming—from the fundamental bare metal programming to the bootloader to the boot-up of an Android system. The focus is on instilling general programming knowledge as well as developing compiler and debugging skills. The objective is to provide basic knowledge about embedded system programming as a good foundation, thereby providing a path to the more advanced areas of embedded system programming.

The book is organized in a very process-oriented way. You can decide how to read this book based on your individual circumstance—that is, in which order to read chapters and explore subtopics. An explanation of how each part of the book relates to the others will help you make this decision.

The book consists of three parts. Part I focuses on so-called bare metal programming, which includes the fundamentals of low-level programming and Android system programming. Chapters 1 through 4 provide essential knowledge related to bare metal programming, including how to run programs on the hardware directly using assembly language code. In Chapter 5, the focus moves to the C programming language. The rest of Part I explores the minimum set of hardware interfaces necessary to boot a Linux kernel using U-Boot. In Chapters 5 to 8, we focus on the hardware interface programming of serial ports, interrupt controllers, the real-time clock, and NAND flash controllers in the bare metal programming environment.

Part II begins with Chapter 9, which covers how to port U-Boot to the goldfish platform. Using U-Boot, we can boot the Linux kernel and Android system, as explained in Chapter 10. The work completed in Chapters 5 through 8 can contribute to the U-Boot porting by isolating the hardware complexity from the driver framework in U-Boot. The same technique can be used in the Linux driver development as well. In Part II, we also use the file system images from the Android SDK to boot the Android system. To support
two different boot processes (NOR and NAND flash), we must customize the file system from the Android SDK. Because this work takes place at the binary level, we are restricted to performing customization at the file level; that is, we cannot change the content of any files. Strategies to customize the file system are covered in Part III.

In Part III, we move from the bootloader to the kernel to the file system. We use a virtual device to demonstrate how to build a customized ROM for an Android device. We explore ways to support a new device and to integrate the bootloader and Linux kernel in the Android source code tree. In Chapter 11, we delve into the environment setup process and the standard build process for the Android emulator. In Chapter 12, we create a customized ROM for the virtual device including the integration of U-Boot and the Linux kernel. At the end of this chapter, readers will have a complete picture just like the Android system developers do at the mobile device manufacturing level.

A detailed introduction to each of the book’s chapters follows. Part I, “Bare Metal Programming” consists of Chapters 1 to 8 focusing on so-called bare metal programming:

- Chapter 1, “Introduction to Embedded System Programming,” gives a general introduction to embedded system programming. It also explains the scope of this book.
- Chapter 2, “Inside Android Emulator,” introduces the Android emulator and gives a brief introduction to the hardware interfaces used throughout the book.
- Chapter 3, “Setting Up the Development Environment,” details the development environment and tools used in our project. It also provides the first example, which gives us a chance to test our environment.
- Chapter 4, “Linker Script and Memory Map,” covers the basics of developing an assembly program. We use two examples to analyze how a program is assembled and linked. After we have a binary image, we analyze how it is loaded into the Android emulator and then started.
- Chapter 5, “Using the C Language,” introduces the C startup code and explains how we move from assembly language to a C language environment. We also begin to explore the goldfish hardware interfaces of the goldfish platform. Likewise, we explore the serial port of the goldfish platform.
- Chapter 6, “Using the C Library,” presents details on how to integrate a C runtime library into a bare metal programming environment. We introduce different flavors of C runtime libraries and use Newlib as an example to illustrate how to integrate a C runtime library.
- Chapter 7, “Exception Handling and Timer,” explores the interrupt controllers, timer, and real-time clock (RTC) of the goldfish platform. We work through various examples that demonstrate ways to handle these hardware interfaces. All example code developed in the chapter can subsequently be used for U-Boot porting in Chapter 9.
- Chapter 8, “NAND Flash Support in Goldfish,” explores the NAND flash interface of the goldfish platform. This is also an important part of U-Boot porting. In Chapter 10, we explore how to boot the Android system from NAND flash.
Part 2, “U-Boot” consists of Chapters 9 and 10, which introduce the processes of U-Boot porting and debugging. After we have a working U-Boot image, we can use it to boot our own goldfish kernel and the Android image.

- Chapter 9, “U-Boot Porting,” gives the details on U-Boot porting.
- Chapter 10, “Using U-Boot to Boot the Goldfish Kernel,” discusses how to build a goldfish Linux kernel on our own. This kernel image is then used to demonstrate the various scenarios to boot the goldfish Linux kernel using U-Boot. Both the NOR flash and NAND flash boot-up processes are discussed.

Part 3, “Android System Integration” considers how to integrate U-Boot and the Linux kernel into the Android Open Source Project (AOSP) and CyanogenMod source trees.

- Chapter 11, “Building Your Own AOSP and CyanogenMod,” gives the details on Android emulator builds in AOSP and CyanogenMod.
- Chapter 12, “Customizing Android and Creating Your Own Android ROM,” teaches you how to create your own Android ROM on a virtual Android device. This Android ROM can be brought up by U-Boot, which we created in Chapter 9.

Example Code

Throughout this book, many examples are available to test the content in each chapter. It is recommended that you input and run the example code while you read this book. Doing so will give you good hands-on experiences and provide you with valuable insight so that you will better understand the topics covered in each chapter.

For Chapters 3 through 8, the directory structure organizes the code by chapter. Some folders are common to all of the examples, such as those containing include and driver files. All other folders are chapter specific, such as c03, c04, and c05; these folders contain the example code in that chapter.

The common makefile is makedefs.arm, which is found in the top-level directory. Individual makefiles are also provided for each example. Following is a template of the makefile for example code. The PROJECTNAME is defined as the filename of an example code. This makefile template is used for the individual projects in Chapters 3 through 8.

```bash
# The base directory relative to this folder
ROOT=../..
PROJECTNAME=

# Include the common make definitions.
include $(ROOT)/makedefs.arm
```
# The default rule, which causes the ${PROJECTNAME} example to be built.
#
all: ${COMPILER}
all: ${COMPILER}/${PROJECTNAME}.axf

#
# The rule to debug the target using Android emulator.
#
debug:
  @ddd --debugger arm-none-eabi-gdb ${COMPILER}/${PROJECTNAME}.axf &
  @emulator -verbose -show-kernel -netfast -avd hd2 -shell -qemu -monitor telnet::6666.server -s -S -kernel ${COMPILER}/${PROJECTNAME}.axf

#
# The rule to clean out all the build products.
#
clean:
  @rm -rf ${COMPILER} ${wildcard *~}

#
# The rule to create the target directory.
#
${COMPILER}:
  @mkdir -p ${COMPILER}

#
# Rules for building the ${PROJECTNAME} example.
#
${COMPILER}/${PROJECTNAME}.axf: ${COMPILER}/${PROJECTNAME}.o
${COMPILER}/${PROJECTNAME}.axf: ${PROJECTNAME}.ld
SCATTERgcc_${PROJECTNAME}=${PROJECTNAME}.ld
ENTRY_${PROJECTNAME}=ResetISR

#
# Include the automatically generated dependency files.
#
ifeq (${MAKECMDGOALS},clean)
  -include ${wildcard ${COMPILER}/*.d} __dummy__
endif
The rest of source code in this book can be found on GitHub at https://github.com/shugaoye/. Please refer to Appendix A for the details.

Conventions Used in This Book
The following typographical conventions are used in this book:

- *Italic* indicates URLs.
- <!-- Bold in angle brackets --> is used to signify comments in the code or console output.
- Constant-width type is used for program listings, as well as within paragraphs to refer to program elements such as variable and function names, databases, data types, environment variables, statements, and keywords.
- Constant-width bold type shows commands or other text that should be typed in by the user.
- Constant-width italic type shows text that should be replaced with the user-supplied values or with the values determined by the context.

Note

A Note signifies a tip, suggestion, or general note.
I am grateful to Laura Lewin and Bernard Goodwin, both executive editors at Pearson Technology Group, who gave me the opportunity to publish this book with Addison-Wesley. I would like to thank the team from Addison-Wesley. Michael Thurston was the developmental editor; he reviewed all the chapters and gave me valuable suggestions on the content presentation. Olivia Basegio and Michelle Housley helped me to coordinate with the team at Addison-Wesley. Project editor Elizabeth Ryan ensured that this project adhered to schedule. I would also like to thank the copy editor, Jill Hobbs, who did a great job improving the readability of this book.

This book could not have been published without technical review. I would like to thank all of the reviewers for identifying errors and for providing valuable feedback about the content. Thanks are especially due to the Android experts, Zigurd Mednieks and G. Blake Meike. They are co-authors of Android-related books, including Enterprise Android and Programming Android.

I also want to thank all of my friends and colleagues at Motorola and Emerson. We had a wonderful time working on many great products that contributed to the technology boom that has occurred in the past 10 years. Together, we witnessed the introduction of the high-tech products that have changed our lives today.

Last but not least, I would like to thank my dearest wife and my lovely daughter, who gave me lots of support and encouragement along the way while I worked on this book.
Roger Ye is an embedded system programmer who has great interest in embedded systems and the latest technologies related to them. He has worked for Motorola, Emerson, and Intel as an engineering manager. At Motorola and Emerson, he was involved in embedded system projects for mobile devices and telecommunications infrastructures. He is now an engineering manager at Intel Security, leading a team that develops Android applications.

Roger now lives in China with his wife Bo Quan and his daughter Yuxin Ye. You can find more information about him at GitHub: https://github.com/shugaoye/.
This page intentionally left blank
Once we have U-Boot ready for the goldfish platform, we can use it to boot the Linux kernel in the Android emulator. Ideally, the boot process starts from nonvolatile memory (such as flash memory). Many kind of storage devices can be used in an embedded system, though NOR and NAND flash devices are the most popular options. In this chapter, we will build a goldfish Linux kernel first. We then explore how to boot Android from NOR flash and NAND flash using U-Boot and this kernel.

Building the Goldfish Kernel

Ideally, we might like to build everything on our own—from the bootloader, to the kernel, to the file system. Except for Google-specific applications, everything in Android is hosted in a project called Android Open Source Project (AOSP). However, we will lose our focus if we go into too much detail about every aspect of the build process right now. We will discuss AOSP builds in Part III of this book. If you want to learn how to build AOSP from scratch, the book *Embedded Android* by Karim Yaghmour is a good reference. In addition, the Internet provides plenty of articles that explain how to work on AOSP.

To build the kernel, we need two things: a prebuilt toolchain and goldfish kernel source code. The recommended option is to use the prebuilt toolchain from AOSP, which can be downloaded from the Google source git repository. Other prebuilt toolchains can be used as well. For example, we could use a prebuilt toolchain from a vendor such as Mentor Graphics (i.e., Sourcery CodeBench).

If you already have an AOSP source tree, you can use the prebuilt toolchain from AOSP directly. If you don’t have an AOSP source tree, the instructions in this chapter explain how to download this toolchain. If you installed your toolchain using the script `install.sh` introduced in Appendix A, you should have the toolchain from CodeBench Lite. In this case, you can skip the steps for downloading AOSP toolchain given in this chapter.

We will use the file system included with the Android SDK to boot up our kernel. When an Android virtual device is created, a corresponding file system is created as well. We will use the virtual device `hd2` that we created in Chapter 2 in this chapter. The file system image for `hd2` can be found at `~/android/avd/hd2.avd`.
You might wonder why we want to build the kernel ourselves instead of using the original kernel in the Android SDK to demonstrate the boot-up process. The reason is that we may not be able to boot up the Linux kernel as smoothly as we think. Actually, this process will most likely fail when we first attempt it. Thus we need a debug build to debug the boot process.

The porting of U-Boot actually includes two steps. First, we must add the necessary hardware support so that we can run U-Boot until the command-line prompt becomes available. Second, we must change U-Boot to prepare the proper environment for the Linux kernel so that control can be transferred to the kernel and the kernel can be started normally. In the second step, if we don’t have a debug version of kernel, it will be very difficult for us to debug U-Boot itself. We will demonstrate how to debug both U-Boot and the Linux kernel at the source code level in this chapter.

### Prebuilt Toolchain and Kernel Source Code

The latest information about how to build an Android kernel using a prebuilt toolchain can be found at https://source.android.com/. Given that AOSP changes from time to time, be aware that the procedure in this chapter is what was available at the time of this book’s writing—and that a newer version may have been released since then.

You can download the prebuilt toolchain from the AOSP git repository using the following command:

```bash
$ git clone https://android.googlesource.com/platform/prebuilts/gcc/linux-x86/arm/arm-eabi-4.7
```

It may take a while for this command to complete its work. After the prebuilt toolchain is downloaded, we can set up the path environment variable to include it:

```bash
$ export PATH=$(pwd)/arm-eabi-4.7/bin:$PATH
```

The next step is to get the goldfish kernel source code. We can use the following command to get a copy of kernel from AOSP repository:

```bash
$ git clone https://android.googlesource.com/kernel/goldfish.git
$ cd goldfish
$ git branch -a
* master
remotes/origin/HEAD -> origin/master
remotes/origin/android-goldfish-2.6.29
remotes/origin/android-goldfish-3.4
remotes/origin/linux-goldfish-3.0-wip
remotes/origin/master
$ git checkout -t origin/android-goldfish-2.6.29 -b goldfish
```
To build the kernel, use the following commands:

```
$ export ARCH=arm
$ export SUBARCH=arm
$ export CROSS_COMPILE=arm-eabi-
$ make goldfish_armv7_defconfig
$ make
```

After the build is completed, we have a release build of the goldfish kernel by default.

To debug the kernel, we need to turn on debugging options in the kernel configuration file. To do so, we can either edit the `.config` file directly or run `menuconfig`. To run `menuconfig`, you have to install the package `libncurses5-dev` first, if you haven’t already installed it:

```
$ sudo apt-get install libncurses5-dev
$ make menuconfig CROSS_COMPILE=arm-eabi-
```

After `menuconfig` starts, we can select `Kernel hacking`, `Compile the kernel with debug info`, as shown in Figure 10.1. An alternative approach, as mentioned earlier, is to edit the `.config` file directly. In the `.config` file, we can set `CONFIG_DEBUG_INFO=y`.

![Figure 10.1 Enabling debugging in menuconfig](image)
Even though these steps look quite simple, problems may occasionally occur. Yet another alternative is to follow the instructions in Appendix B to set up the development environment and build everything in this book using the `Makefile` and scripts in the repository `build` in GitHub.

**Running and Debugging the Kernel in the Emulator**

After the build process is finished, we can run and debug the kernel in the Android emulator. The compressed kernel image can be found at `arch/arm/boot/zImage`. This image can be used to run the kernel in the emulator. The image file `vmlinux` is in ELF format; it can be used by gdb to get the debug symbol. We give the following command to start the Android emulator using our own kernel image:

```
$ emulator -verbose -show-kernel -netfast -avd hd2 -qemu -serial stdio -s -S -kernel arch/arm/boot/zImage
```

After the emulator is running, we can start the gdb debugger to debug the kernel. We will use the graphical interface ddd to start the gdb debugger; it produces a more user-friendly environment. In the following command line, we tell ddd to use `arm-eabi-gdb` as the debugger and `vmlinux` as the binary image:

```
$ ddd --debugger arm-eabi-gdb vmlinux
```

After gdb starts, it needs to connect to the gdb server in the emulator using the command `target remote localhost:1234`. To track the boot-up progress, we can set a breakpoint at the function `start_kernel`:

```
GNU DDD 3.3.12 (i686-pc-linux-gnu), by Dorothea Lütkehaus and Andreas Zeller.  
Copyright © 1995-1999 Technische Universität Braunschweig, Germany.  
Copyright © 1999-2001 Universität Passau, Germany.  
Copyright © 2001 Universität des Saarlandes, Germany.  
Copyright © 2001-2004 Free Software Foundation, Inc.  
Reading symbols from /home/sgye/src/Android/goldfish/vmlinux...done.  
(gdb) target remote localhost:1234  
0x00000000 in ?? ()  
(gdb) b start_kernel  
Breakpoint 1 at 0xc0008858: file init/main.c, line 531.  
(gdb) c
```
After starting the process, gdb will stop at `start_kernel`, as shown in Figure 10.2.
After the system boots up, a console like that shown in Figure 10.3 appears. The entire Android system should be ready to use at this point. From here, we boot the kernel in the Android emulator directly. In the next few sections, we will boot this kernel using U-Boot.
Booting Android from NOR Flash

QEMU doesn’t provide NOR flash emulation on the goldfish platform. To make things simple, we will use RAM to create a boot-up process that is similar to the boot process from NOR flash. This approach builds a binary image that includes U-Boot, the Linux kernel, and the RAMDISK image and passes this image to QEMU through the `–kernel` option.

Before we start, let’s look at how QEMU boots a Linux kernel. To boot up a Linux kernel, the bootloader prepares the following environment:

- The processor is in SVC (Supervisor) mode and IRQ and FIQ are disabled.
- MMU is disabled.
- Register r0 is set to 0.
- Register r1 contains the ARM Linux machine type.
- Register r2 contains the address of the kernel parameter list.

After power-up, QEMU starts to run from address 0x00000000. Before it loads a kernel image, QEMU prepares the environment described previously; it then jumps to address 0x00010000. Figure 10.4 shows a memory dump before the point at which QEMU launches a kernel image. Notice the five lines of assembly code before control is transferred to the kernel image—these lines are hard-coded by QEMU when the system starts. The first line (0x00000000) sets register r0 to 0. The second line (0x00000004) and third line (0x00000008) set register r1 to 0x5a1, which is the machine type of the goldfish

Figure 10.3  Linux console after boot-up

```bash
<6>yaffs: dev is /dev/rdsk/c2t1l0  name is "mtdblock2"
yaffs: dev is /dev/rdsk/c2t1l0  name is "mtdblock2"
<6>yaffs: passed flags ""
yaffs: Attempting MTD mount on 31:2, "mtdblock2"
yaffs: Attempting MTD mount on 31:2, "mtdblock2"
yaffs_read_super: isCheckpointed 0
yaffs_read_super: isCheckpointed 0
<3>init: cannot find '/system/etc/install-recovery.sh', disabling 'flash_recovery'
init: cannot find '/system/etc/install-recovery.sh', disabling 'flash_recovery'
<3>init: untracked pid 47 exited
init: untracked pid 47 exited
<6>warning: 'rild' uses 32-bit capabilities (legacy support in use)
warning: 'rild' uses 32-bit capabilities (legacy support in use)
<6>eth0: link up
eth0: link up
shell@android:/ $ <7>eth0: no IPv6 routers present
<6>request_suspend_state: wakeup (3->0) at 35350000414 (2013-05-25 13:53:51.16129604 UTC)
request_suspend_state: wakeup (3->0) at 35350000414 (2013-05-25 13:53:51.16129604 UTC)
shell@android:/ $  
```
platform. The fourth line (0x0000000c) sets the value of register r2 to 0x100, which is the start address of the kernel parameter list. The fifth line (0x00000010) sets the register pc to 0x10000, so the execution jumps to address 0x10000. QEMU assumes the kernel image is loaded at address 0x10000.

As outlined in Figure 10.5, we will create an image including U-Boot, the Linux kernel, and RAMDISK for testing. U-Boot is located at address 0x00010000, which is the address that QEMU will invoke. The Linux kernel is located at address 0x00210000, and the RAMDISK image is located at address 0x00410000. Both the kernel and RAMDISK images are placed at a distance of 2MB starting from address 0x00010000. After U-Boot is relocated, it will move itself to address 0x1f59000 (this address may change for each build) and free about 2MB from the starting address 0x00010000. We can inform U-Boot about the kernel and RAMDISK image locations through the bootm command, given
from the U-Boot command line. Alternatively, you can set the default `bootm` parameter in `include/configs/goldfish.h`. We can add the default `bootm` and kernel parameters in `goldfish.h` as follows:

```
#define CONFIG_BOOTARGS "qemu.gles=1 qemu=1 console=ttyS0 android.qemud=ttys1
androidboot.console=ttys2 android.checkjni=1 ndns=1"
#define CONFIG_BOOTCOMMAND "bootm 0x210000 0x410000"
```

The U-Boot command `bootm` then copies the kernel image into 0x00010000 and the RAMDISK image into 0x00800000. At that point, U-Boot jumps to address 0x00010000 to start the Linux kernel.

**Creating the RAMDISK Image**

Besides U-Boot and the kernel image, we need a RAMDISK image to support the boot process. In Android, RAMDISK is used as the root file system. We can customize the boot
Booting Android from NOR Flash

Let's create a RAMDISK image so that we can build the flash image for testing. Given that we are using the Android emulator, we can take advantage of the RAMDISK image from the Android SDK as the base for our image. The RAMDISK image can be found in the system image folder in the Android SDK. For example, the RAMDISK image for Android 4.0.3 (API 15) can be found at {Android SDK installation path}/system-images/android-15/armeabi-v7a/ramdisk.img.

If we want to modify this image, we can create a folder and extract the image to that folder using the following command:

```bash
$ mkdir initrd
$ cd initrd
$ gzip -dc < ../ramdisk.img | cpio --extract
```

Once we extract the RAMDISK image, we can see its content:

```bash
$ ls -F
data/  dev/  init.goldfish.rc*  proc/  sys/  ueventd.goldfish.rc
default.prop  init*  init.rc*  sbin/  system/  ueventd.rc
```

The RAMDISK includes the folders and startup scripts for the root file system. The actual system files are stored in system.img, and the user data files are stored in userdata.img. Both system.img and userdata.img are emulated as NAND flash. They are mounted as /system and /data folders, respectively, under the root file system.

We can inspect file systems after boot-up as follows:

```bash
shell@android:/ $ mount
rootfs / rootfs ro 0 0
tmpfs /dev tmpfs rw,nosuid,nodev,mode=755 0 0
devpts /dev/pts devpts rw,mode=600 0 0
proc /proc proc rw 0 0
sysfs /sys sysfs rw 0 0
none /acct cgroup rw,cpuacct 0 0
tmpfs /mnt/secure tmpfs rw,mode=700 0 0
tmpfs /mnt/asec tmpfs rw,mode=755,gid=1000 0 0
tmpfs /mnt/obb tmpfs rw,mode=755,gid=1000 0 0
none /dev/cpuctl cgroup rw,cpu 0 0/dev/block/mtdblock0 /system yaffs2 ro 0 0
/dev/block/mtdblock1 /data yaffs2 rw,nosuid,nodev 0 0
/dev/block/mtdblock2 /cache yaffs2 rw,nosuid,nodev 0 0
shell@android:/ $
```
Now we can change the files in this folder as desired. After we’ve made those changes, we can generate the new RAMDISK image using the following commands:

```bash
$ find . > ../initrd.list
$ cpio -o -H newc -O ../ramdisk.img < ../initrd.list
$ cd..
$ gzip ramdisk.img
$ mv ramdisk.img.gz rootfs.img
```

### Creating the Flash Image

Now that all of the image files (U-Boot, Linux kernel, and RAMDISK) are ready, we can start to create the flash image to boot the system.

U-Boot can boot a variety of file types (e.g., ELF, BIN), but these file types have to first be repackaged in the U-Boot image format (i.e., uImage). This format stores information about the operating system type, the load address, the entry point, basic integrity verification (via CRC), compression types, free description text, and so on.

To create a U-Boot image format, we need a utility called `mkimage`. If this tool is not installed in the host system, it can be installed in Ubuntu using the following command:

```bash
$ sudo apt-get install uboot-mkimage
```

With this utility, we can repackage the kernel image and RAMDISK image in the U-Boot format using the following commands:

```bash
$ mkimage -A arm -C none -O linux -T kernel -d zImage -a 0x00010000 -e 0x00010000 zImage.uimg
$ gzip -c rootfs.img > rootfs.img.gz
$ mkimage -A arm -C none -O linux -T ramdisk -d rootfs.img.gz -a 0x00800000 -e 0x00800000 rootfs.uimg
```

Once we have uImage files in hand, we can generate a flash image using the `dd` command as follows:

```bash
$ dd if=/dev/zero of= flash.bin bs=1 count=6M
$ dd if=u-boot.bin of= flash.bin conv=notrunc bs=1
$ dd if= zImage.uimg of= flash.bin conv=notrunc bs=1 seek=2M
$ dd if= rootfs.uimg of= flash.bin conv=notrunc bs=1 seek=4M
```

The file `flash.bin` includes all three images that we will use to boot up the system.

There are multiple steps to build the Linux kernel and generate all images. Please refer to Appendix A for the detailed procedures. All related Makefiles and scripts can be found in repository `build` in GitHub.

### Booting Up the Flash Image

Finally, we are ready to boot the flash image that we built. Let’s run it in the Android emulator and stop in the U-Boot command-line interface first. In U-Boot, we set a
2-second delay before U-Boot starts autoboot. Before autoboot starts, any keystroke will take us to the U-Boot command prompt. We can use a U-Boot command to verify the kernel and RAMDISK image, thereby making sure they are correct:

```
$ emulator -verbose -show-kernel -netfast -avd hd2 -qemu -serial stdio -kernel flash.bin
```

U-Boot 2013.01.-rc1-00001-g54217a1 (Feb 09 2014 - 23:28:59)

U-Boot code: 00010000 -> 00029B0C  BSS: -> 0002D36C
IRQ Stack: 0badc0de
FIQ Stack: 0badc0de
monitor len: 0001D36C
ramsize: 20000000
TLB table at: 1ff00000
Top of RAM usable for U-Boot at: 1ff00000
Reserving 116k for U-Boot at: 1ff62000
Reserving 136k for malloc() at: 1fff0000
Reserving 32 Bytes for Board Info at: 1ffaffde
Reserving 120 Bytes for Global Data at: 1ffaff68
Reserving 8192 Bytes for IRQ stack at: 1ffadf68
New Stack Pointer is: 1ffad5f8

RAM Configuration:
Bank #0: 00000000 512 MiB
relocation Offset is: 1ffc2000
goldfish_init(), gtty.base=ff012000
WARNING: Caches not enabled
monitor flash len: 0001D0D4
Now running in RAM - U-Boot at: 1ff62000
Using default environment

Destroy Hash Table: 1ffeb724 table = 00000000
Create Hash Table: N=89
INSERT: table 1ffeb724, filled 1/89 rv 1ffb02a4 ==> name="bootargs" value="qemu.gles=1 qemu=1 console=ttys0 android.qemud=ttys1 androidboot.console=ttys2 android.checkjni=1 ndns=1"
INSERT: table 1ffeb724, filled 2/89 rv 1ffb0160 ==> name="bootcmd" value="bootm 0x210000 0x410000"
INSERT: table 1ffeb724, filled 3/89 rv 1ffb02f8 ==> name="bootdelay" value="2"
INSERT: table 1ffeb724, filled 4/89 rv 1ffb0178 ==> name="baudrate" value="38400"
Chapter 10 Using U-Boot to Boot the Goldfish Kernel

In the preceding code, notice that we use the `iminfo` command to check the image at 0x00210000 and 0x00410000. U-Boot recognizes the data at these addresses as the Linux kernel image and Linux RAMDISK image, respectively. Also notice the load
address: U-Boot loads the kernel image to address 0x00010000 and the RAMDISK image to address 0x00800000.

We can boot the system using the `bootm` command as follows:

```
Goldfish # bootm 0x210000 0x410000
```

```
## Current stack ends at 0xffadb10 * kernel: cmdline image address = 0x00210000
## Booting kernel from Legacy Image at 00210000 ...

  Image Name:
  Image Type:  ARM Linux Kernel Image (uncompressed)
  Data Size:   1722596 Bytes = 1.6 MiB
  Load Address: 00010000
  Entry Point: 00010000
  kernel data at 0x00210040, len = 0x001a48e4 (1722596)

* ramdisk: cmdline image address = 0x00410000
## Loading init Ramdisk from Legacy Image at 00410000 ...

  Image Name:
  Image Type:  ARM Linux RAMDisk Image (uncompressed)
  Data Size:   187687 Bytes = 183.3 KiB
  Load Address: 00800000
  Entry Point: 00800000
  ramdisk start = 0x00800000, ramdisk end = 0x0082dd27
  Loading Kernel Image ... OK

CACHE: Misaligned operation at range [00010000, 006a2390]
OK
  kernel loaded at 0x00010000, end = 0x001b48e4
using: ATAGS
## Transferring control to Linux (at address 00010000)...

Starting kernel ...

Uncompressing Linux............................................................
........................................... done, booting the kernel.
goldfish_fb_get_pixel_format:167: display surface,pixel format:
  bits/pixel: 16
  bytes/pixel: 2
  depth: 16
  red: bits=5 mask=0xf800 shift=11 max=0x1f
  green: bits=6 mask=0x7e0 shift=5 max=0x3f
Chapter 10  Using U-Boot to Boot the Goldfish Kernel

blue:        bits=5 mask=0x1f shift=0 max=0x1f
alpha:       bits=0 mask=0x0 shift=0 max=0x0
Initializing cgroup subsys cpu
Linux version 2.6.29-ge3d684d (sgye@sgye-Latitude-E6510) (gcc version 4.6.3 (Sourcery CodeBench Lite 2012.03-57) ) #1 Sun Feb 9 23:32:29 CST 2014
CPU: ARMv7 Processor [410fc080] revision 0 (ARMv7), cr=10c5387f
CPU: VIPT nonaliasing data cache, VIPT nonaliasing instruction cache
Machine: Goldfish
Memory policy: ECC disabled, Data cache writeback
Built 1 zonelists in Zone order, mobility grouping on.  Total pages: 130048
Kernel command line: qemu.gles=1 qemu=1 console=ttyS0 android.qemud=ttys1 androidboot.console=ttys2 android.checkjni=1 ndns=1
Unknown boot option 'qemu.gles=1': ignoring
Unknown boot option 'android.qemud=ttys1': ignoring
Unknown boot option 'androidboot.console=ttys2': ignoring
Unknown boot option 'android.checkjni=1': ignoring
PID hash table entries: 2048 (order: 11, 8192 bytes)
Console: colour dummy device 80x30
Dentry cache hash table entries: 65536 (order: 6, 262144 bytes)
Inode-cache hash table entries: 32768 (order: 5, 131072 bytes)
Memory: 512MB = 512MB total
Memory: 515456KB available (2944K code, 707K data, 124K init)
Calibrating delay loop... 370.27 BogoMIPS (lpj=1851392)
Mount-cache hash table entries: 512
Initializing cgroup subsys debug
Initializing cgroup subsys cpuacct
Initializing cgroup subsys freezer
CPU: Testing write buffer coherency: ok
net_namespace: 936 bytes
NET: Registered protocol family 16
bio: create slab <bio-0> at 0
NET: Registered protocol family 2
IP route cache hash table entries: 16384 (order: 4, 65536 bytes)
TCP established hash table entries: 65536 (order: 7, 524288 bytes)
TCP bind hash table entries: 65536 (order: 6, 262144 bytes)
TCP: Hash tables configured (established 65536 bind 65536)
TCP reno registered

NET: Registered protocol family 1

checking if image is initramfs... it is

Freeing initrd memory: 180K

goldfish_new_pdev goldfish_interrupt_controller at ff000000 irq -1

goldfish_new_pdev goldfish_device_bus at ff001000 irq 1

goldfish_new_pdev goldfish_timer at ff002000 irq 3

goldfish_new_pdev goldfish_rtc at ff010000 irq 10

goldfish_new_pdev goldfish_tty at ff002000 irq 4

goldfish_new_pdev goldfish_tty at ff011000 irq 11

goldfish_new_pdev goldfish_tty at ff012000 irq 12

goldfish_new_pdev smc91x at ff013000 irq 13

goldfish_new_pdev goldfish_fb at ff014000 irq 14

goldfish_new_pdev goldfish_audio at ff004000 irq 15

goldfish_new_pdev goldfish_mmc at ff005000 irq 16

goldfish_new_pdev goldfish_memlog at ff006000 irq -1

goldfish_new_pdev goldfish_battery at ff015000 irq 17

goldfish_new_pdev goldfish_events at ff016000 irq 18

goldfish_new_pdev goldfish_nand at ff017000 irq -1

goldfish_new_pdev qemu_pipe at ff018000 irq 19

goldfish_new_pdev goldfish-switch at ff01a000 irq 20

goldfish_new_pdev goldfish-switch at ff01b000 irq 21

goldfish_pdev_worker registered goldfish_interrupt_controller

goldfish_pdev_worker registered goldfish_device_bus

goldfish_pdev_worker registered goldfish_timer

goldfish_pdev_worker registered goldfish_rtc

goldfish_pdev_worker registered goldfish_tty

goldfish_pdev_worker registered goldfish_tty

goldfish_pdev_worker registered goldfish_tty

goldfish_pdev_worker registered smc91x

goldfish_pdev_worker registered goldfish_fb

goldfish_pdev_worker registered goldfish_audio

goldfish_pdev_worker registered goldfish_mmc

goldfish_pdev_worker registered goldfish_memlog

goldfish_pdev_worker registered goldfish_battery
goldfish_pdev_worker registered goldfish_events
goldfish_pdev_worker registered goldfish_nand
goldfish_pdev_worker registered qemu_pipe
goldfish_pdev_worker registered goldfish-switch
goldfish_pdev_worker registered goldfish-switch
ashmem: initialized
Installing knfsd (copyright (C) 1996 okir@monad.swb.de).
fuse init (API version 7.11)
yaffs Feb 9 2014 23:30:30 Installing.
msgmni has been set to 1007
alg: No test for stdrng (krng)
io scheduler noop registered
io scheduler anticipatory registered (default)
io scheduler deadline registered
io scheduler cfq registered
allocating frame buffer 480 * 800, got ffa00000
console [ttyS0] enabled
brd: module loaded
loop: module loaded
nbd: registered device at major 43
goldfish_audio_probe
tun: Universal TUN/TAP device driver, 1.6
tun: (C) 1999-2004 Max Krasnyansky <maxk@qualcomm.com>
smc91x.c: v1.1, sep 22 2004 by Nicolas Pitre <nico@cam.org>
eth0 (smc91x): not using net_device_ops yet
eth0: SMC91C11xD (rev 1) at e080c000 IRQ 13 [nowait]
eth0: Ethernet addr: 52:54:00:12:34:56
goldfish nand dev0: size c5e0000, page 2048, extra 64, erase 131072
goldfish nand dev1: size c200000, page 2048, extra 64, erase 131072
goldfish nand dev2: size 4000000, page 2048, extra 64, erase 131072
mice: PS/2 mouse device common for all mice
*** events probe ***
events_probe() addr=0xe0814000 irq=18
events_probe() keymap=qwerty2
input: qwerty2 as /devices/virtual/input/input0
goldfish rtc goldfish rtc: rtc core: registered goldfish rtc as rtc0
device-mapper: uevent: version 1.0.3
logger: created 64K log 'log_main'
logger: created 256K log 'log_events'
logger: created 64K log 'log_radio'
Netfilter messages via NETLINK v0.30.
nf_conntrack version 0.5.0 (8192 buckets, 32768 max)
CONFIG_NF_CT_ACCT is deprecated and will be removed soon. Please use
nf_conntrack.acct=1 kernel parameter, acct=1 nf_conntrack module option or
sysctl net.netfilter.nf_conntrack_acct=1 to enable it.
ctnetlink v0.93: registering with nfnetlink.
NF_TPROXY: Transparent proxy support initialized, version 4.1.0
NF_TPROXY: Copyright (c) 2006-2007 BalaBit IT Ltd.
xt_time: kernel timezone is -0000
ip_tables: (C) 2000-2006 Netfilter Core Team
arp_tables: (C) 2002 David S. Miller
TCP cubic registered
NET: Registered protocol family 10
ip6_tables: (C) 2000-2006 Netfilter Core Team
IPv6 over IPv4 tunneling driver
NET: Registered protocol family 17
NET: Registered protocol family 15
RPC: Registered udp transport module.
RPC: Registered tcp transport module.
802.1Q VLAN Support v1.8 Ben Greear <greearb@candleatech.com>
All bugs added by David S. Miller <davem@redhat.com>
VFP support v0.3: implementor 41 architecture 3 part 30 variant c rev 0
goldfish rtc goldfish rtc: setting system clock to 2014-02-20 08:54:53 UTC
(1392886493)
Freeing init memory: 124K
mmc0: new SD card at address e118
mmcblk0: mmc0:e118 SU02G 100 MiB
mmcblk0:
init: cannot open '/initlogo.rle'
yaffs: dev is 32505856 name is "mtdblock0"
Source-Level Debugging of the Flash Image

At this point, we can use a flash image that includes both U-Boot and the goldfish kernel to boot up the system. But can we do source-level debugging as well? If we are working on a real hardware board with JTAG debugger, it is quite difficult to do source-level debugging for both U-Boot and the kernel. However, no such problem arises in a virtual environment. With this approach, we can closely observe the transition from bootloader to Linux kernel using source-level debugging. This is a convenient way to debug the U-Boot boot-up process. We can track the interaction between U-Boot and Linux kernel by tracing the execution of the source code.

Let's start the Android emulator with gdb support:

```
$ emulator -verbose -show-kernel -netfast -avd hd2 -shell -qemu -s -S -kernel flash.bin
```

We connect to the Android emulator using gdb:

```
$ ddd --debugger arm-none-eabi-gdb u-boot/u-boot
```

As shown in Figure 10.6, we load U-Boot in gdb with source-level debugging information.

Now we can perform source-level debugging for U-Boot. Since U-Boot will reload itself, we must use the same technique that we applied in Chapter 9 to continue the source-level debugging after memory relocation occurs.

Each time we start U-Boot in gdb, we have to go through a series of steps. It is much easier (and faster) to put these steps into a gdb script, as shown in Example 10.1. This script can be found in the folder `bin` of the repository `build`.
Example 10.1. GDB Startup Script for U-Boot (u-boot.gdb)

```bash
# Debug u-boot
b board_init_f

c
b relocate_code
c
p/x ((gd_t *)$r1)->relocaddr
d
symbol-file ./u-boot/u-boot
add-symbol-file ./u-boot/u-boot 0x1ff59000
b board_init_r
```
We can load this script in the gdb console using the following command:

```
(gdb) target remote localhost:1234
(gdb) source bin/u-boot.gdb
```

After running this script, we can see that U-Boot has stopped at `board_init_f()` and the U-Boot symbol has been reloaded to the memory address after its relocation, as shown in Figure 10.7.

Let's continue running U-Boot to a point after memory relocation. In the script `u-boot.gdb`, the breakpoint is set to `board_init_r()`. After U-Boot stops at this breakpoint, we can load the goldfish kernel symbol. The multiple steps to load the goldfish kernel can also be put into a gdb script, as shown in Example 10.2. This script can also be found in the folder `bin` of the repository `build`.

![Figure 10.7 Reload the U-Boot symbol after relocation](image)

**Example 10.2 GDB Script for Debugging Goldfish Kernel (goldfish.gdb)**

```
# Debug goldfish kernel

d
symbol-file ./goldfish/vmlinux
add-symbol-file ./goldfish/vmlinux 0x00010000
b start_kernel
```
We can load the script goldfish.gdb to the gdb console as follows:

```plaintext
(gdb) source bin/goldfish.gdb
add symbol table from file "/home/sgye/src/build/goldfish/vmlinux" at
    .text_addr = 0x10000
Breakpoint 4 at 0xc00086b4: file /home/sgye/src/goldfish/init/main.c, line 535.
(2 locations)
...
warning: (Internal error: pc 0x10088 in read in psymtab, but not in symtab.)
(gdb) c
warning: (Internal error: pc 0x10088 in read in psymtab, but not in symtab.)
Breakpoint 4, start_kernel () at /home/sgye/src/goldfish/init/main.c:535
(gdb)
```

In the script goldfish.gdb, the kernel symbol is loaded from vmlinux at memory address 0x10000 and a breakpoint is set at `start_kernel()`. After loading the kernel symbol, we can continue running U-Boot. Now the system stops at the Linux kernel code, as shown in Figure 10.8.

![Figure 10.8 The goldfish kernel at start_kernel()](image-url)
As we can see in this session, we have much more control over the system in the virtual environment compared to what is possible in the real hardware. In turn, we can perform a deeper analysis of the code by tracing the execution path at the source level. We can work at the source level, starting from the first line of code and working all the way to the point at which the operating system fully boots up.

**Booting Android from NAND Flash**

With U-Boot, we can also boot Android from NAND flash. This approach is very similar to that used in real-world cases. When using this approach, we keep everything (kernel, RAMDISK image, and file system) in NAND flash and boot from there. As discussed in Chapter 8, three flash devices—system, userdata, and cache—are connected to the Android emulator. Even though Android mounts the RAMDISK as root, all system files are included in system.img. We can put both the kernel and RAMDISK images in system.img as well, allowing us to then boot the entire system from system.img.

**Preparing system.img**

To put the kernel and RAMDISK images into system.img, we have to recreate them. As mentioned previously, in Android 4.3 and earlier, system.img is in the YAFFS2 format. In Android 4.4 or later, it is in the ext4 format. In the ext4 format, we can mount the system.img file directly and copy both the kernel and RAMDISK in it. In this chapter, we will continue to use the Android Virtual Device hd2 that we created in Chapter 2; it is in YAFFS2 format and relies on Android version 4.0.3.

To regenerate system.img, we need to use YAFFS2 utilities. You can get them after you check out the build repository from GitHub. Two utilities—mkyaffs2image and unyaffs—can be found in the bin folder. Their source code can be found at http://code.google.com.

We have to extract system.img first. After we extract it, we can copy the kernel and RAMDISK images to the system image folder. As we did in the previous section, we need them in the U-Boot format (zImage.uimg and rootfs.uimg).

We can regenerate system.img using the mkyaffs2image command:

```bash
$ mkdir system
$ cd system
$ unyaffs ../system.img
$ cd ..
$ cp ./rootfs.uimg system/ramdisk.uimg
$ cp ./zImage.uimg system/zImage.uimg
$ rm ./system.img
$ mkyaffs2image system ./system.img
```

Now we have a new system.img that contains both the kernel and RAMDISK images. We can use it to boot Android with U-Boot. For the exact procedures, refer to the build target rootfs of Makefile in the build repository.
Booting from NAND Flash

To boot Android from NAND flash, we need to use the `-system` option to tell the emulator to use our version of `system.img` instead of the one that comes with the Android SDK:

```
$ emulator -show-kernel -netfast -avd hd2 -shell -system ./system.img -ramdisk ./ramdisk.img -qemu -kernel ./u-boot.bin
```

```
U-Boot 2013.01.-rc1-00005-g4627a3e-dirty (Mar 07 2014 - 15:55:45)
U-Boot code: 00010000 -> 0006E2BC BSS: -> 000A6450
IRQ Stack: 0badc0de
FIQ Stack: 0badc0de
monitor len: 00096450
ramsize: 20000000
TLB table at: 1ff00000
Top of RAM usable for U-Boot at: 1ff0000
Reserving 601k for U-Boot at: 1ff59000
Reserving 4104k for malloc() at: 1fb57000
Reserving 32 Bytes for Board Info at: 1fb56fe0
Reserving 120 Bytes for Global Data at: 1fb56f68
Reserving 8192 Bytes for IRQ stack at: 1fb54f68
New Stack Pointer is: 1fb54f58
RAM Configuration:
Bank #0: 00000000 512 MiB
relocation Offset is: 1ff49000
goldfish_init(), getty.base=ff012000
WARNING: Caches not enabled
monitor flash len: 00065AD4
Now running in RAM - U-Boot at: 1ff59000
NAND: base=ff017000
goldfish_nand_init: id=0: name=nand0, nand_name=system
goldfish_nand_init: id=1: name=nand1, nand_name=userdata
goldfish_nand_init: id=2: name=nand2, nand_name=cache
459 MiB
Using default environment
```
Chapter 10  Using U-Boot to Boot the Goldfish Kernel

Destroy Hash Table: 1ffb5fe4 table = 00000000
Create Hash Table: N=89
INSERT: table 1ffb5fe4, filled 1/89 rv 1fb572a4 ==> name="bootargs" value="qemu.gles=1 qemu=1 console=ttys0 android.qemud=ttys1 android.boot.console=ttys2 android.checkjni=1 ndns=1"
INSERT: table 1ffb5fe4, filled 2/89 rv 1fb57160 ==> name="bootcmd" value="bootm 0x210000 0x410000"
INSERT: table 1ffb5fe4, filled 3/89 rv 1fb571f8 ==> name="bootdelay" value="2"
INSERT: table 1ffb5fe4, filled 4/89 rv 1fb57178 ==> name="baudrate" value="38400"
INSERT: table 1ffb5fe4, filled 5/89 rv 1fb57154 ==> name="bootfile" value="/tftpboot/uImage"
INSERT: free(data = 1fb57008)
INSERT: done
In:    serial
Out:   serial
Err:   serial
Net:   SMC91111-0
Warning: SMC91111-0 using MAC address from net device
### main_loop entered: bootdelay=2
### main_loop: bootcmd="bootm 0x210000 0x410000"
Hit any key to stop autoboot:  0
## Current stack ends at 0x1fb54b00  * kernel: cmdline image address = 0x00210000
Wrong Image Format for bootm command
ERROR: can't get kernel image!
Command failed, result=1
Goldfish #

After the emulator is running, we are sent to the U-Boot command prompt because we have interrupted the autoboot process. We can then mount system.img from the U-Boot command line. First, we use the U-Boot command ydevconfig to configure the NAND device. We configure the device name as sys starting from block 0 to 0x64d (1613). The device number is 0:

Goldfish # ydevconfig sys 0 0x0 0x64d
Configures yaffs mount sys: dev 0 start block 0, end block 1613

We can check the configuration using the command ydevls:

Goldfish # ydevls
sys 0 0x00000 0x0064d not mounted
Next, we use the `ymount` command to mount the device `sys`. After mounting the device, we can list its contents using the command `yls`:

```
Goldfish # ymount sys
Mounting yaffs2 mount point sys
Goldfish # yls sys
build.prop
media
fonts
lib
ramdisk.uimg
usr
zImage.uimg
xbin
etc
framework
tts
bin
app
lost+found
```

Once we find both the kernel and RAMDISK image (`zImage.uimg` and `ramdisk.uimg`), we need to load them into memory using the command `yrdm` before we can boot the system. After we load them into memory, we can use the command `iminfo` to verify them:

```
Goldfish # yrdm sys/ramdisk.uimg 0x410000
Copy sys/ramdisk.uimg to 0x00410000...     [DONE]
Goldfish # iminfo 0x410000
## Checking Image at 00410000 ...
  Legacy image found
  Image Name: ARM Linux RAMDisk Image (uncompressed)
  Data Size: 187703 Bytes = 183.3 KiB
  Load Address: 00800000
  Entry Point: 00800000
  Verifying Checksum ... OK
Goldfish # yrdm sys/zImage.uimg 0x210000
```
Chapter 10  Using U-Boot to Boot the Goldfish Kernel

Copy sys/zImage.uimg to 0x00210000...  [DONE]

Goldfish # **iminfo 0x210000**

## Checking Image at 00210000 ...
  Legacy image found
  Image Name: ARM Linux Kernel Image (uncompressed)
  Data Size: 1722852 Bytes = 1.6 MiB
  Load Address: 00010000
  Entry Point: 00010000
  Verifying Checksum ... OK

Now we are ready to boot the system. This stage is the same as what we did when booting with NOR flash in the previous section. We use the **umount** command to dismount the YAFFS2 file system first and use the **bootm** command to boot the system:

Goldfish # **yumount sys**

Unmounting yaffs2 mount point sys

Goldfish # **bootm 0x2100000 0x410000**

## Current stack ends at 0x1fb54b10 *  kernel: cmdline image address = 0x00210000

## Booting kernel from Legacy Image at 00210000 ...

  Image Name: ARM Linux Kernel Image (uncompressed)
  Data Size: 1722852 Bytes = 1.6 MiB
  Load Address: 00010000
  Entry Point: 00010000
  kernel data at 0x00210040, len = 0x001a49e4 (1722852)
  *  ramdisk: cmdline image address = 0x00410000

## Loading init Ramdisk from Legacy Image at 00410000 ...

  Image Name: ARM Linux RAMDisk Image (uncompressed)
  Data Size: 187703 Bytes = 183.3 KiB
  Load Address: 00800000
  Entry Point: 00800000
  ramdisk start = 0x00010000, ramdisk end = 0x0082dd37

Loading Kernel Image ... OK

CACHE: Misaligned operation at range [00010000, 006a2790]
Booting Android from NAND Flash

OK

kernel loaded at 0x00010000, end = 0x001b49e4
using: ATAGS

## Transferring control to Linux (at address 00010000)...

Starting kernel ... 

Uncompressing Linux.................................
................. done, booting the kernel.

goldfish_fb_get_pixel_format:167: display surface, pixel format:
  bits/pixel: 16
  bytes/pixel: 2
  depth: 16
  red:    bits=5 mask=0xf800 shift=11 max=0x1f
  green:  bits=6 mask=0x7e0 shift=5 max=0x3f
  blue:   bits=5 mask=0x1f shift=0 max=0x1f
  alpha:  bits=0 mask=0x0 shift=0 max=0x0

Initializing cgroup subsys cpu

Linux version 2.6.29-ge3d684d (sye1@ubuntu) (gcc version 4.6.3 (Sourcery CodeBench Lite 2012.03-57) ) #4 Fri Mar 7 15:59:39 CST 2014
CPU: ARMv7 Processor [410fc080] revision 0 (ARMv7), cr=10c5387f
CPU: VIPT nonaliasing data cache, VIPT nonaliasing instruction cache
Machine: Goldfish
Memory policy: ECC disabled, Data cache writeback
Built 1 zonelists in Zone order, mobility grouping on. Total pages: 130048
Kernel command line: qemu.gles=1 qemu=1 console=ttyS0 android.qemud=ttyS1 androidboot.console=ttyS2 android.checkjni=1 ndns=1
Unknown boot option 'qemu.gles=1': ignoring
Unknown boot option 'android.qemud=ttyS1': ignoring
Unknown boot option 'androidboot.console=ttyS2': ignoring
Unknown boot option 'android.checkjni=1': ignoring
PID hash table entries: 2048 (order: 11, 8192 bytes)
Console: colour dummy device 80x30
Dentry cache hash table entries: 65536 (order: 6, 262144 bytes)
Inode-cache hash table entries: 32768 (order: 5, 131072 bytes)
Memory: 512MB = 512MB total
Memory: 515456KB available (2956K code, 707K data, 124K init)
Chapter 10 Using U-Boot to Boot the Goldfish Kernel

Calibrating delay loop... 452.19 BogoMIPS (lpj=2260992)
Mount-cache hash table entries: 512
Initializing cgroup subsys debug
Initializing cgroup subsys cpucacct
Initializing cgroup subsys freezer
CPU: Testing write buffer coherency: ok
net_namespace: 936 bytes
NET: Registered protocol family 16
bio: create slab <bio-0> at 0
NET: Registered protocol family 2
IP route cache hash table entries: 16384 (order: 4, 65536 bytes)
TCP established hash table entries: 65536 (order: 7, 524288 bytes)
TCP bind hash table entries: 65536 (order: 6, 262144 bytes)
TCP: Hash tables configured (established 65536 bind 65536)
TCP reno registered
NET: Registered protocol family 1
checking if image is initramfs... it is
Freeing initrd memory: 180K
goldfish_new_pdev goldfish_interrupt_controller at ff000000 irq -1
goldfish_new_pdev goldfish_device_bus at ff001000 irq 1
goldfish_new_pdev goldfish_timer at ff003000 irq 3
goldfish_new_pdev goldfish_rtc at ff001000 irq 10
goldfish_new_pdev goldfish_tty at ff002000 irq 4
goldfish_new_pdev goldfish_tty at ff011000 irq 11
goldfish_new_pdev goldfish_tty at ff012000 irq 12
goldfish_new_pdev smc91x at ff013000 irq 13
goldfish_new_pdev goldfish_fb at ff014000 irq 14
goldfish_new_pdev goldfish_audio at ff004000 irq 15
goldfish_new_pdev goldfish_mmc at ff005000 irq 16
goldfish_new_pdev goldfish_memlog at ff006000 irq -1
goldfish_new_pdev goldfish_battery at ff015000 irq 17
goldfish_new_pdev goldfish_events at ff016000 irq 18
goldfish_new_pdev goldfish_nand at ff017000 irq -1
goldfish_new_pdev qemu_pipe at ff018000 irq 19
goldfish_new_pdev goldfish-switch at ff01a000 irq 20
goldfish_new_pdev goldfish-switch at ff01b000 irq 21
goldfish_pdev_worker registered goldfish_interrupt_controller
goldfish_pdev_worker registered goldfish_device_bus
goldfish_pdev_worker registered goldfish_timer
goldfish_pdev_worker registered goldfish_RTC
goldfish_pdev_worker registered goldfish_IRQ
goldfish_pdev_worker registered goldfish_IRQ
goldfish_pdev_worker registered smc91x
goldfish_pdev_worker registered goldfish_FB
goldfish_pdev_worker registered goldfish_AUDIO
goldfish_pdev_worker registered goldfishMMC
goldfish_pdev_worker registered goldfish_MEMLOG
goldfish_pdev_worker registered goldfish_EVENTS
goldfish_pdev_worker registered goldfish_BATTERY
goldfish_pdev_worker registered goldfish_NAND
goldfish_pdev_worker registered qemu_PIPE
goldfish_pdev_worker registered goldfish-SWITCH
ashmem: initialized
Installing knfsd (copyright (C) 1996 okir@monad.swb.de).
fuse init (API version 7.11)
msgmni has been set to 1007
alg: No test for stdrng (krng)
io scheduler noop registered
io scheduler anticipatory registered (default)
io scheduler deadline registered
io scheduler cfg registered
allocating frame buffer 480 * 800, got ffa00000
console [ttyS0] enabled
brd: module loaded
loop: module loaded
Chapter 10  Using U-Boot to Boot the Goldfish Kernel

nbd: registered device at major 43

goldfish_audio_probe
tun: Universal TUN/TAP device driver, 1.6
tun: (C) 1999-2004 Max Krasnyansky <maxk@qualcomm.com>

smc91x.c: v1.1, sep 22 2004 by Nicolas Pitre <nico@cam.org>

eth0 (smc91x): not using net_device_ops yet

eth0: SMC91C11xFD (rev 1) at e080c000 IRQ 13 [nowait]

eth0: Ethernet addr: 52:54:00:12:34:56

goldfish nand dev0: size c9c0000, page 2048, extra 64, erase 131072
goldfish nand dev1: size c200000, page 2048, extra 64, erase 131072
goldfish nand dev2: size 4000000, page 2048, extra 64, erase 131072

mice: PS/2 mouse device common for all mice

*** events probe ***

events_probe() addr=0xe0814000 irq=18

input: qwerty2 as /devices/virtual/input/input0
goldfish_rtc goldfish_rtc: rtc core: registered goldfish_rtc as rtc0
device-mapper: uevent: version 1.0.3

logger: created 64K log 'log_main'
logger: created 256K log 'log_events'
logger: created 64K log 'log_radio'

Netfilter messages via NETLINK v0.30.

nf_conntrack version 0.5.0 (8192 buckets, 32768 max)
CONFIG_NF_CT_ACCT is deprecated and will be removed soon. Please use
nf_conntrack.acct=1 kernel parameter, acct=1 nf_conntrack module option or
sysctl net.netfilter.nf_conntrack_acct=1 to enable it.

ctnetlink v0.93: registering with nfnetlink.

NF_TPROXY: Transparent proxy support initialized, version 4.1.0

NF_TPROXY: Copyright (c) 2006-2007 BalaBit IT Ltd.

xt_time: kernel timezone is -0000

ip_tables: (C) 2000-2006 Netfilter Core Team

arp_tables: (C) 2002 David S. Miller

TCP cubic registered
NET: Registered protocol family 10
ip6_tables: (C) 2000-2006 Netfilter Core Team
IPv6 over IPv4 tunneling driver
NET: Registered protocol family 17
NET: Registered protocol family 15
RPC: Registered udp transport module.
RPC: Registered tcp transport module.
802.1Q VLAN Support v1.8 Ben Greear <greearb@candelatech.com>
All bugs added by David S. Miller <davem@redhat.com>
VFP support v0.3: implementor 41 architecture 3 part 30 variant c rev 0
goldfish_rtc goldfish_rtc: setting system clock to 2014-03-10 10:04:08 UTC (1394445848)
Freeing init memory: 124K
mmc0: new SD card at address e118
mmcblk0: mmc0:e118 SU02G 100 MiB
  mmcblk0:
  init: cannot open '/initlogo.rle'
yaffs: dev is 32505856 name is "mtdblock0"
yaffs: passed flags ""
yaffs: Attempting MTD mount on 31.0, "mtdblock0"
yaffs_read_super: isCheckpointed 0
save exit: isCheckpointed 1
yaffs: dev is 32505857 name is "mtdblock1"
yaffs: passed flags ""
yaffs: Attempting MTD mount on 31.1, "mtdblock1"
yaffs_read_super: isCheckpointed 0
yaffs: dev is 32505858 name is "mtdblock2"
yaffs: passed flags ""
yaffs: Attempting MTD mount on 31.2, "mtdblock2"
yaffs_read_super: isCheckpointed 0
init: cannot find '/system/etc/install-recovery.sh', disabling 'flash_recovery'
eth0: link up
shell@android:/ $ warning: 'rild' uses 32-bit capabilities (legacy support in use)
Summary

In this chapter, we used U-Boot to demonstrate two scenarios for operating system boot-up. First, we booted Android from NOR flash using U-Boot. Even though the Android emulator doesn’t have NOR flash, we created an image to simulate it. Second, we booted Android from NAND flash. In this case, we put the kernel and RAMDISK images inside `system.img` and used U-Boot to boot the system.

We can build almost everything on our own to boot the Android system, except RAMDISK and the file system. To make our own RAMDISK and file system, we hacked them from the Android SDK. In next two chapters, we will go even further; that is, we will explore how to build everything, including the Android file system, from source code.
This page intentionally left blank
Index

A

ABI Research, 283
Address alignment, viewing, 31–32
ALARM_HIGH register, 19
ALARM_LOW register, 19
align directive, 44
all, build target, 36
Android
  file system, 183–185
  kernel, verifying, 273–274
  virtual devices, setting up, 284–287
android command, 16
Android emulator
  configuring virtual devices, 14–16
  description, 11
  illustration, 14
  mobile device hardware features supported, 13–14
Android emulator, building with AOSP
  AOSP build environment, 288–289
  building Android emulator images, 290–292
  downloading AOSP source, 289
  initializing a repo client, 289
  installing repo tool, 289
  installing required packages, 288
  installing the JDK, 288
  testing AOSP images, 292–297
Android emulator, building with CyanogenMod
  armemu virtual device, 332–338, 353–354
  build process, 298–302
  downloading CyanogenMod source, 297
  emulator images, Android version of, 306
  emulator images, building, 298–302
  emulator images, testing, 302–307
  introduction, 297
  releases, 284
Android ROM, creating with AOSP
  booting Android with U-Boot from NAND flash, 323–332
  building the kernel, 317–322
  building U-Boot, 322–323
  process description, 309–317
Android ROM, creating with CyanogenMod
  booting CyanogenMod, 337–338
  building the kernel, 334–337
Android ROM, creating with
CyanogenMod (continued)
building U-Boot, 337–338
introduction, 332–334
Android SDK, setting up, 27, 284–287
Android Virtual Device Manager, 284–287
AOSP (Android Open Source Project). See also
CyanogenMod.
armemu virtual device, 352–353
introduction, 283–284
releases, 284
AOSP (Android Open Source Project), Android emulator
build
AOSP build environment, 288–289
downloading AOSP source, 289
initializing a repo client, 289
installing required packages, 288
installing the JDK, 288
testing AOSP images, 292–297
AOSP (Android Open Source Project), creating Android
ROM
booting Android with U-Boot from NAND flash, 323–332
building the kernel, 317–322
building U-Boot, 322–323
process description, 309–317
AOSP (Android Open Source Project), supporting new
hardware
booting Android with U-Boot from NAND flash, 323–332
building the kernel, 317–322
building U-Boot, 322–323
process description, 309–317
AOSP-based smartphones, sales growth, 283
APCS registers use convention, 78
Application layer, embedded systems, 7
Architecture of embedded systems, 7–10
ARM Architectural Reference Manual, 5
ARM processors for embedded systems, 8–9. See also
specific processors.
ARM register set, 67
ARM System Developer’s Guide, 5, 125
ARM926EJ-S processor, 9
ARM_dAbort() function, 137
armemu virtual device
Android version, illustration, 317, 321
building a kernel for, 318–322
building with AOSP, 352–353
building with CyanogenMod, 353–354
creating, 284–287, 309–317
creating skeleton files for, 310–311
CyanogenMod version, 332–338
arm_exc.S file, 141, 142–149
ARM_fiq() function, 137
ARM_irq() function, 137
ARM_pAbort() function, 137
ARM_reserved() function, 137
ARM_swi() function, 137
ARM_undefined() function, 137
Assembler directives, 30, 43–44
Assembly initialization phase, 103
Assembly language, 30
AVD Manager, 16. See also SDK Manager.
AVDs (Android Virtual Devices), 14–16

B
Bare metal programming. See also Embedded system
programming.
common programming languages, 4
definition, 3, 5
resources for, 5
__BARE_METAL__ macro, 191
Barr, Michael, 5
Ben-Yossef, Gilad, 5
Binary files, 31–32
Bionic, C library variant, 95
board_early_init_f() function, 234
board_init() function, 234
board_nand_init() function, 205
Books and publications. See also Online resources.
ARM Architectural Reference Manual, 5
ARM System Developer's Guide, 5, 125
Building Embedded Linux Systems, 5
Embedded Android, 5, 249, 288
Linkers and Loaders, 42
Procedure Call Standard for the ARM Architecture, 78
Programming Embedded System in C and C++, 5
RealView Compilation Tools Developer Guide, 50
RealView Platform Baseboard, 5
Booting
CyanogenMod, 337–338
flash image, 258–266
a Linux kernel, 254
Booting Android from NAND flash
boot process, 271–279
checking the configuration, 272
introduction, 270
preparing system.img, 270
with U-Boot, 323–332
verifying the kernel and the RAMDISK image, 273–274
Booting Android from NOR flash
introduction, 254–256
memory relocation, 256
RAMDISK image, creating, 256–258
Booting Android from NOR flash, flash image
booting, 258–266
creating, 258
source-level debugging, 266–270
Booting the goldfish kernel
booting a Linux kernel, 254
building the kernel, 249–250
debugging the kernel, 252–254
kernel source code, 250–252
prebuilt toolchain, 250–252
running the kernel, 252–254
Bootloader. See U-Boot.

bootm command, 256, 261, 274

BSP_irq() function, 149

.bss section
C language in a bare metal environment, 68–78
eroaming our, 72–78

bss segment, 42
Building Embedded Linux Systems, 5

.byte directive, 43–44

BYTES_READY register, 18

C functions, calling from assembly language code, 79–81

C language in a bare metal environment. See also C functions.

.bss section, 68–78
.data section, 68–78
.global directive, 81
.isr_vector section, 68–78
.rodata section, 68–78, 80
.stack section, 68–78
.text section, 68–78
calling assembly language functions from, 81
calling C functions from assembly language code, 79–81
calling convention, 78–81
debugging, 75–76
global variables, 68
preparing the stack, 65–68
prerequisites for, 63–65
read-only data, 68
startup code, 68–78
version information, 80
viewing symbol placement, 76–78

C language in a bare metal environment, example code
calling C code from assembly language code, 64–65
linker script, 70–72

C library variants. See also Newlib C library:

Semihosting support.
in a bare metal environment, 94–96
Bionic, 95
debugging capabilities, 95
libcmtd.lib, 94
libcmu.lib, 94
Microsoft C runtime libraries, 94
msvcrt.lib, 94
msvctrd.lib, 94
msvcr71.lib, 94
RealView Development Suite, 95
uclibc, 95

C Run-Time (CRT) libraries, 94
c04e1.s file, 42–46
c04e2.c file, 53–56
c04e3.c file, 56–57
c04e4.c file, 58
c05e1.c file, 64–65
c05e1.1d file, 64, 70–72
c05e2.c file, 81, 90–91
c06e1.c file, 97, 113–115
c06e2.1d file, 97–103
c06e2.c file, 118–122
c07e1.c file, 128, 134–137
c07e1.1d file, 128
c07e2.c file, 141, 152–155
c07e2.1d file, 141
c07e3.c file, 174
c08e1.c file, 211–216
c08e1.1d file, 211–216

calling
assembly language functions from C, 81
C convention, 78–81
clean, build target, 36
CLEAR_ALARM register, 19
CLEAR_INTERRUPT register, 19
Client development environment, setting up, 25–26

CMD register, 18

CMD_INT_DISABLE command, 18
CMD_INT_ENABLE command, 18
CMD_READ_BUFFER command, 18
CMD_WRITE_BUFFER command, 18


Coding. See Programming.

Commands

android, 16
bootm, 256, 261, 274

CMD_INT_DISABLE, 18
CMD_INT_ENABLE, 18
CMD_READ_BUFFER, 18
CMD_WRITE_BUFFER, 18
d, 138, 179
e, 138, 179
g, 179
git-diff, 221

iminfo, 260, 273

ld, 31–32

mount, 184–185

NAND_CMD_BLOCK_BAD_SET, 187
NAND_CMD_ERASE, 187
NAND_CMD_GET_DEV_NAME Cơ, 187
NAND_CMD_READ, 187
NAND_CMD_WRITE, 187

nm, 31–32

objcopy, 32
q, 91
r, 179
s, 179
t, 138, 179

umount, 274
x, 179

Comments, assembly language, 30

Common files, folder for, 81

Compiling U-Boot, 220–224

CONFIG_USE_IRQ macro, 239

Console
debugging serial ports, console log example, 91
interrupts, enabling/disabling, 18
Constant names, 69
Copying
  converting file formats, 32
  .data to RAM, 57–58, 68, 72–78
  objcopy command, 32
CORTEX-A processors, 8–9
CORTEX-M processors, 8–9
CORTEX-R processors, 8–9
CRT (C Run-Time) libraries, 94
Customizing Android, supporting new hardware with AOSP
  booting Android with U-Boot from NAND flash, 323–332
  building the kernel, 317–322
  building U-Boot, 322–323
  process description, 309–317
Customizing Android, supporting new hardware with CyanogenMod
  booting CyanogenMod, 337–338
  building the kernel, 334–337
  building U-Boot, 337–338
  introduction, 332–334
CyanogenMod, Android emulator build. See also AOSP (Android Open Source Project)
  armemu virtual device, 332–338, 353–354
  build process, 298–302
  downloading CyanogenMod source, 297
  emulator images, Android version of, 306
  emulator images, building, 298–302
  emulator images, testing, 302–307
  introduction, 297
  releases, 284
CyanogenMod, creating Android ROM
  booting CyanogenMod, 337–338
  building the kernel, 334–337
  building U-Boot, 337–338
  introduction, 332–334
CyanogenMod, supporting new hardware
  booting CyanogenMod, 337–338
  building the kernel, 334–337
  building U-Boot, 337–338
  introduction, 332–334
CyanogenMod wiki, 332

d command, 138, 179
Data buffer, checking, 87–88
  .data section
    copying to RAM, 57–58, 68, 72–78
    placement, 68–78
Data segment, 42
DATA_LEN register, 18
DATA_PTR register, 18
Date and time
  getting/setting, 179
  resetting, 173
date.c file, 174
ddd
  installing, 29
  starting, 33
  starting GDB in, 37
  stepping through instructions, 35
  user interface, 35–36
  viewing register contents, 34
debug, build target, 37
DEBUG option, 37–38
Debugging
  C library variant capabilities, 95
  development environment, 25, 37–38
  the goldfish kernel, 252–254
  makefile template, 37–38
  quitting, 91
  serial ports, console log example, 91
  source-level, 75–76
  source-level flash image, 266–270
  U-Boot, with GDB, 224–227
Debugging, GDB (GNU Debugger)
  scripts for, 227, 267–270
Denx, Wolfgang, 220
Development environment, debugging, 25, 37–38. See also Virtualization environment.
Development environment, setting up
  Android SDK, 27
  building the binary, 30–32
  for client, 25–26
  downloading/installing toolchains, 26–29
  flash memory, emulating, 32–36
  for host, 25–26
  makefile template, build targets, 32–36
  output filename, specifying, 31
  running in the Android emulator, 32–36
Device drivers
  adding drivers, 239
  adding to goldfish, 239
  device driver changes, 239–246
  Ethernet drivers, 245
  NAND flash drivers, 241–243
  RTC drivers, 243–245
  serial drivers, 239–241
Disabling. See Enabling/disabling.
Dollar sign ($), in assembly language labels, 30
Downloading
  EABI/ELF toolchain, 28–29
  git repositories, 356–357
  GNU/Linux toolchain, 28–29
  Sourcery CodeBench Lite, 28–29
  Sourcery CodeBench trial version, 28–29
  toolchains, 26–29
  U-Boot, 220–224
dram_init() function, 234
Drivers. See Device drivers.

E

e command, 138, 179
EABI/ELF toolchain, downloading, 28–29
Eclipse editor, 29
ED (empty descending) stacks, 66
Editors, 29
Index

Files, serial_goldfish.c (continued)
NAND flash test program, 210–216
Newlib C library, 96
unit test of timer and RTC, 174
Files, startup.s
calling C code in assembler language, 72–77
calling main from, 80
goldfish serial support, 81
Files, syscallcalls.c
nested interrupt handler, 141
Newlib C library, 96, 105–112
a simple interrupt handler, 128
unit test of timer and RTC, 174
Files, timer.c
description, 128
NAND flash test program, 210–216
nested interrupt handler, 140
timer interface functions, 166–171
unit test of timer and RTC, 174
Flash devices. See NAND flash.
Flash image
booting, 258–266
creating, 258
Flash memory. See also NAND flash; NOR flash.
copying .data to RAM, 57–58, 68
definition, 39
eulating, 32–36
specifying a file for, 32–36
4byte directive, 43–44
FPGA Cores processors, 8–9
FTL (Flash Translation Layer), 188
Full ascending (FA) stacks, 66
Full descending (FD) stacks, 66
Functions
ARM_dAbort() function, 137
ARM_fiq() function, 137
ARM_irq() function, 137
ARM_pAbort() function, 137
ARM_reserved() function, 137
ARM_swi() function, 137
ARM_undef() function, 137
assembly language, calling from C language, 81
board_early_init_f(), 234
board_init(), 234
board_nand_init(), 205
BSP_irq(), 149
C language, calling from assembly language, 79–81
dram_init(), 234
EnterUserMode(), 150–152
goldfish_disable_all_irq(), 131
goldfish_getc(), 88–89
goldfish_get_irq_num() function, 131
goldfish_gets() function, 88–89
goldfish_irq_status(), 132
goldfish_mask_irq(), 131
goldfish_nand_cmd(), 205–206
goldfish_nand_erase(), 206
goldfish_nand_init_device(), 205
goldfish_nand_isbad(), 206
goldfish_nand_markbad(), 206
goldfish_nand_read(), 206
goldfish_nand_read_oob(), 206
goldfish_nand_write(), 206
goldfish_nand_write_oob(), 206
goldfish_putchar(), 88–89
goldfish_unmask_irq(), 131
interrupt support, 128–132
misc_init_r(), 234
mktime(), 174
rtc_get(), 174
rtc_reset(), 174
rtc_set(), 174
sw_handler(), 152–155
SystemCall(), 150–152
timer_init(), 172
to_tm(), 174
__udelay(), 172
void goldfish_clear_timer(), 172
void goldfish_set_timer(), 172
G

G command, 179
GDB (GNU Debugger)
scripts for, 227, 267–270
starting in ddd, 37
gdb command prompt, 81
Gerum, Philippe, 5
get_millisecond() function, 172
goldfish clear_timer(), 172
goldfish_set_timer(), 172
Gerrum, Philippe, 5
goldfish disable all_irq(), 131
goldfish enable irq(), 131
goldfish getc(), 88–89
goldfish get irq num(), 131
goldfish gets(), 88–89
goldfish irq status(), 132
goldfish_mask irq(), 131
goldfish nand cmd(), 205–206
goldfish nand erase(), 206
goldfish nand init device(), 205

Files, serial_goldfish.c (continued)
NAND flash test program, 210–216
Newlib C library, 96
unit test of timer and RTC, 174
Files, startup.s
calling C code in assembler language, 72–77
calling main from, 80
goldfish serial support, 81
Files, syscallcalls.c
nested interrupt handler, 141
Newlib C library, 96, 105–112
a simple interrupt handler, 128
unit test of timer and RTC, 174
Files, timer.c
description, 128
NAND flash test program, 210–216
nested interrupt handler, 140
timer interface functions, 166–171
unit test of timer and RTC, 174
Flash devices. See NAND flash.
Flash image
booting, 258–266
creating, 258
Flash memory. See also NAND flash; NOR flash.
copying .data to RAM, 57–58, 68
definition, 39
eulating, 32–36
specifying a file for, 32–36
4byte directive, 43–44
FPGA Cores processors, 8–9
FTL (Flash Translation Layer), 188
Full ascending (FA) stacks, 66
Full descending (FD) stacks, 66
Functions
ARM_dAbort() function, 137
ARM_fiq() function, 137
ARM_irq() function, 137
ARM_pAbort() function, 137
ARM_reserved() function, 137
ARM_swi() function, 137
ARM_undef() function, 137
assembly language, calling from C language, 81
board_early_init_f(), 234
board_init(), 234
board_nand_init(), 205
BSP_irq(), 149
C language, calling from assembly language, 79–81
dram_init(), 234
EnterUserMode(), 150–152
goldfish_disable_all_irq(), 131
goldfish_getc(), 88–89
goldfish_get_irq_num() function, 131
goldfish_gets() function, 88–89
goldfish_irq_status() function, 132
goldfish_mask_irq() function, 131
goldfish_nand_cmd() function, 205–206
goldfish_nand_erase() function, 206
goldfish_nand_init_device() function, 205
goldfish_nand_isbad() function, 206
goldfish_nand_markbad() function, 206
goldfish_nand_read() function, 206
goldfish_nand_read_oob() function, 206
goldfish_nand_write() function, 206
goldfish_nand_write_oob() function, 206
goldfish_putchar() function, 88–89
goldfish_unmask_irq() function, 131
interrupt support, 128–132
misc_init_r() function, 234
mktime() function, 174
rtc_get() function, 174
rtc_reset() function, 174
rtc_set() function, 174
sw_handler() function, 152–155
SystemCall() function, 150–152
timer_init() function, 172
to_tm() function, 174
__udelay() function, 172
void goldfish_clear_timer() function, 172
void goldfish_set_timer() function, 172

G command, 179
GDB (GNU Debugger)
scripts for, 227, 267–270
starting in ddd, 37
gdb command prompt, 81
Gerum, Philippe, 5
goldfish getc() function, 88–89
goldfish get irq num() function, 131
goldfish read() function, 206
goldfish read oob() function, 206
goldfish write() function, 206
goldfish write oob() function, 206
GNU toolchain, 11
GNU/Linux toolchain, 28–29
goldfish interrupt controller. See Interrupt controller.
goldfish interrupt handler. See Interrupt handler.
goldfish kernel
source code, URL for, 13
startup log, example code, 19–24
goldfish kernel, initializing
hardware interfaces, example code, 21–22
memory, example code, 20
NAND flash, example code, 22–24
goldfish platform
hardware diagram, 15
serial ports, 18, 81–87
goldfish.c file, 229, 235–239
goldfish disable all_irq() function, 131
goldfish get() function, 205–206
goldfish getc() function, 88–89
goldfish getc() function, 88–89
Git, xvii
git repositories, downloading, 356–357
git-diff command, 221
GitHub, xvii
.global directive, 81
Global variables, 68
GNU toolchain, 11
GNU/Linux toolchain, 28–29
goldfish interrupt controller. See Interrupt controller.
goldfish interrupt handler. See Interrupt handler.
goldfish kernel
source code, URL for, 13
startup log, example code, 19–24
goldfish kernel, initializing
hardware interfaces, example code, 21–22
memory, example code, 20
NAND flash, example code, 22–24
goldfish platform
hardware diagram, 15
serial ports, 18, 81–87
goldfish.c file, 229, 235–239
goldfish disable all_irq() function, 131
goldfish get() function, 205–206
goldfish getc() function, 88–89

Index 365

# Goldfish.h File
- Adding Ethernet drivers, 242
- Description, 229–234
- NAND flash drivers, adding to goldfish, 242
- Serial drivers, adding to goldfish, 240
- **GOLDFISH_INTERRUPT_DISABLE** register, 127
- **GOLDFISH_INTERRUPT_DISABLE_ALL** register, 127
- **GOLDFISH_INTERRUPT_ENABLE** register, 127
- **GOLDFISH_INTERRUPT_NUMBER** register, 127
- **GOLDFISH_INTERRUPT_STATUS** register, 127
- goldfish_irq_status() function, 132
- goldfish_mask_irq() function, 131
- goldfish_nand.c file
  - NAND flash driver, 189, 195–205
  - NAND flash test program, 211–216
- goldfish_nand_cmd() function, 205–206
- goldfish_nand_erase() function, 206
- goldfish_nand_init_device() function, 205
- goldfish_nand_isbad() function, 206
- goldfish_nand_markbad() function, 206
- goldfish_nand_read() function, 206
- goldfish_nand_read_oob() function, 206
- goldfish_nand_reg.h file, 189, 190–191
- goldfish_nand_write() function, 206
- goldfish_nand_write_oob() function, 206
- goldfish_putchar() function, 88–89
- goldfish_uart.c file, 174
- goldfish_uart.S file
  - goldfish serial port support, 81–83
  - NAND flash test program, 210–216
  - Newlib C library, 96
  - Simple interrupt handler, 128, 141
- goldfish_unmask_irq() function, 131
- Google Android SDK, 284–287

**H**

# Hardware
- New, supporting. See Supporting new hardware.
  - Programming directly on. See Bare metal programming.
  - Virtual. See Virtual hardware.
- Hardware interfaces
  - Initializing, example code, 21–22
  - Registers and interrupts, 17
  - Supported by Android emulator, 17–18
- Hardware platform, overview, 11
- hardware.h file, 85–87
- “Hello World” program, 29–30
- Host development environment, setting up, 25–26

# Hardware Reset Phase, 103

# Interrupts
- Console, enabling/disabling, 18
- For hardware interfaces, 17
- Enabling, 142–149
- Example code files, 140–141. See also specific files.
- Implementation, 142–149
- Processor modes, changing, 150–152
- Processor mode switch, discovering, 155–163
- Program status register, 157
- Setting breakpoints, 158–163
- Software interrupt, triggering, 150–152
- Stack pointer addresses, 156
- Stack structure, 156
- Testing, 155–163

# Interrupt Handler, Simplest Form
- Alarms, testing, 139
- Current pending interrupt number, getting, 131
- Enabling/disabling interrupts, 131
- Example code files, 128. See also specific files.
- Implementation, 142–149
- Processor modes, changing, 150–152
- Processor mode switch, discovering, 155–163
- Program status register, 157
- Setting breakpoints, 158–163
- Software interrupt, triggering, 150–152
- Stack pointer addresses, 156
- Stack structure, 156
- Testing, 134–140
- Timer interrupt, testing, 140

# Interrupt Handler, Nested
- Enabling, 142–149
- Example code files, 140–141. See also specific files.
- Implementation, 142–149
- Processor modes, changing, 150–152
- Processor mode switch, discovering, 155–163
- Program status register, 157
- Setting breakpoints, 158–163
- Software interrupt, triggering, 150–152
- Stack pointer addresses, 156
- Stack structure, 156
- Testing, 155–163

# Interrupt Handler, Simplest Form
- Alarms, testing, 139
- Current pending interrupt number, getting, 131
- Enabling/disabling interrupts, 131
- Example code files, 128
- Implementation, 132–134
- Interrupt support functions, 129–132
- Number of current pending interrupts, getting, 132
- Serial input/output, testing, 139
- Startup code, 132–134
- Testing, 134–140
- Timer interrupt, testing, 140

# Integrated Kernel Building,” 334

# Interrupt Controller, 126–128. See also Interrupt handler.

# Interrupt Handler
- Current pending interrupt number, getting, 131
- Enabling/disabling interrupts, 131
- Example code files, 128. See also specific files.
- Implementing, 132–134
- Interrupt support functions, 129–132
- Number of current pending interrupts, getting, 132
- Serial input/output, testing, 139
- Startup code, 132–134
- Testing, 134–140
- Timer interrupt, testing, 140

# Interrupt Handler, Nested
- Enabling, 142–149
- Example code files, 140–141. See also specific files.
- Implementation, 142–149
- Processor modes, changing, 150–152
- Processor mode switch, discovering, 155–163
- Program status register, 157
- Setting breakpoints, 158–163
- Software interrupt, triggering, 150–152
- Stack pointer addresses, 156
- Stack structure, 156
- Testing, 155–163

# Interrupt Handler, Simplest Form
- Alarms, testing, 139
- Current pending interrupt number, getting, 131
- Enabling/disabling interrupts, 131
- Example code files, 128
- Implementing, 132–134
- Interrupt support functions, 129–132
- Number of current pending interrupts, getting, 132
- Serial input/output, testing, 139
- Startup code, 132–134
- Testing, 134–140
- Timer interrupt, testing, 140

# Hardware Interfaces
- Initializing, example code, 21–22
- Registers and interrupts, 17
- Supported by Android emulator, 17–18

# Hardware Platform, Overview, 11

# Hardware.h File
- 85–87
- “Hello World” Program, 29–30

# Host Development Environment, Setting Up, 25–26

# I

# Iminfo Command, 260, 273
- _init System Call, 112

# Initializing
- Hardware interfaces, example code, 21–22
- Memory, example code, 20
- NAND flash, example code, 22–24

# Initializing, Data in RAM
- Accessing Memory, 60–61
- Copying .data to RAM, 57–58, 68
- Example code, 56–57, 58–59
- LMA (Load Memory Address), 57–58
- Load address, 57–58
- Overview, 56–58
- Runtime address, 57
- VMA (Virtual Memory Address), 57

# Input/Output. See also Reading/Writing.
- Serial, Testing, 139
- Serial Ports, 88–89
- Serial Goldfish.c File, 88–89

# Installing
- Ddd, 29
- JDK, 288
- Qemu-system-arm command, 223
- Repo Tool, 289
- Required packages for AOSP, 288
- Toolchains, 26–29

# “Integrated Kernel Building,” 334

# Interrupt Controller, 126–128. See also Interrupt Handler.

# Interrupt Handler
- Current pending interrupt number, getting, 131
- Enabling/disabling interrupts, 131
- Example code files, 128. See also specific files.
- Implementing, 132–134
- Interrupt support functions, 129–132
- Number of current pending interrupts, getting, 132
- Serial input/output, testing, 139
- Startup code, 132–134
- Testing, 134–140
- Timer interrupt, testing, 140

# Interrupt Handler, Nested
- Enabling, 142–149
- Example code files, 140–141. See also specific files.
- Implementation, 142–149
- Processor modes, changing, 150–152
- Processor mode switch, discovering, 155–163
- Processor modes, 157
- Program status register, 157
- Setting breakpoints, 158–163
- Software interrupt, triggering, 150–152
- Stack pointer addresses, 156
- Stack structure, 156
- Testing, 155–163

# Interrupt Handler, Simplest Form
- Alarms, testing, 139
- Current pending interrupt number, getting, 131
- Enabling/disabling interrupts, 131
- Example code files, 128
- Implementation, 132–134
- Interrupt support functions, 129–132
- Number of current pending interrupts, getting, 132
- Serial input/output, testing, 139
- Startup code, 132–134
- Testing, 134–140
- Timer interrupt, testing, 140

# Interrupts
- Console, enabling/disabling, 18
- For hardware interfaces, 17
Index

isr.c file
NAND flash test program, 211–216
a simple interrupt handler, 141, 142–149
unit test of timer and RTC, 174
isr_vector section, 68–78

K
Kaufmann, Morgan, 125
Kernel
Android, verifying, 273–274
building for armemu virtual device, 318–322
building with AOSP, 317–322
building with CyanogenMod, 334–337
“Integrated Kernel Building,” 334
verifying, 273–274
Kernel, Android ROM
supporting new hardware with AOSP, 317–322
supporting new hardware with CyanogenMod, 334–337
Kernel, goldfish
booting a Linux kernel, 254
building, 249–250
building the kernel, 249–250
debugging, 252–254
Linux, booting, 254
prebuilt toolchain, 250–252
running, 252–254
source code, 250–252
startup, example code, 19–24
Kernel, Linux
booting, 254
testing flash info from, 206–210
kernel-qemu image file, 183–184

L
Labels, assembly language, 30
ld command, 31–32
"Learn about the Repo Tool..." 355
Learning embedded system programming, 6
Levine, John R., 42
libcmtd.1ib runtime library, 94
libcnt.1ib runtime library, 94
Linux, history of, 220
Linker. See also Linker script.
2byte directive, 43–44
4byte directive, 43–44
align directive, 44
assembler directives, 43–44
bs segment, 42
byte directive, 43–44
Data segment, 42
definition, 41
description, 41–42
example code, 42–43, 44–46, 46–49
executable output, 42
relocatable code, 50
relocation, 46–49
section merging, 49–50
section placement, 50–51
symbol resolution, 42–43
Text segment, 42
word directive, 43–44
Linker script. See also Linker.
.(period), location counter, 52
*(asterisk), wildcard character, 52
for C language, example code, 70–72
description, 51–53
example code, 53–56
Linkers and Loaders, 42
Linking, 41. See also Linker; Linker script.
Linux
GNU/Linux toolchain, 28–29
starting SDK Manager under, 15
Linux kernel
booting, 254
testing flash info from, 206–210
LMA (load memory address), 57–58
Load address, 57–58
Local manifest file, 356–357
Location counter, period (.), 52, 69
Logs for goldfish kernel startup, example code, 19–24
low_level_init.c file
NAND flash test program, 211–216
a simple interrupt handler, 141
unit test of timer and RTC, 174
lowlevel_init.S file, 229

M
make command, building a development environment, 30–31
makedeabs.arm makefile, template for, xx–xxii
makefile template
build targets, 32–36
debugging, 37–38
example code, xx–xxi
Masters, Jon, 5
Memory
initializing, example code, 20
managing, 112
relocation, booting Android from NOR flash, 256
Memory map, 39–41
Memory Technology Device (MTD). See MTD (Memory Technology Device).
Mentor Graphics, downloading toolchains, 28–29
Microsoft C runtime libraries, 94
Milliseconds/seconds since boot up, getting, 172
misc_init_r() function, 234
MIUI development community, 283–284
mkimage utility, 258
mktime() function, 174
mkvendor.sh script, 332–333
Mobile devices, hardware features supported by Android emulator, 13–14
mount command, 184–185
msvcrt.lib runtime library, 94
msvcmrt.lib runtime library, 94
msvcrtd.lib runtime library, 94
msvcurt.lib runtime library, 94
MTD (Memory Technology Device). See also NAND flash.
compatibility with block devices, 188
NAND flash support for, 188–189
setting up structure for, example code, 203
support for, 188–189
U-Boot API, 205
MTD (Memory Technology Device), example code
cHECKING FOR BAD BLOCKS, 201
data structure, 196
erasing blocks, 197
going/setting bad block data, 206
initializing a device, 202–203
initializing NAND flash controller, 203–204
marking bad blocks, 201–202
**mtd_info** structure, implementing, 190–191
operation sequence, 196–197
reading blocks, 199–200
reading/writing blocks with out-of-band, 198
setting up the MTD structure, 203
writing blocks, 200
mtd.h file, 191–195
**mtd_info** structure, implementing, 190–191
NAND flash. See also Booting Android from NAND flash;
NOR flash.
available storage, calculating, 185
checking for bad blocks, 201
data structure, 196
*eMMC, 184
erasing a page, 185
erasing blocks, 197
FTL (Flash Translation Layer), 188
going/setting bad block data, 206
initializing, example code, 22–24
initializing a device, 202–203
initializing NAND flash controller, 203–204
marking bad blocks, 201–202
MTD (Memory Technology Device) support, 188–189. See also MTD (Memory Technology Device).
**mtd_info** structure, implementing, 190–191
operation sequence, 196–197
*SD/MMC, 184
setting up the MTD structure, 203
*SSD, 184
NAND flash controller
command execution, 188
commands, 187
initializing, 203–204
number of devices connected, detecting, 188
verifying version of, 188
NAND flash device drivers
eXample code, 195–205
functions, 205–206
illustration, 189
NAND flash programming interface
erasing block size, 187
erasing blocks, 206
NAND flash programming interface, device information
capabilities, 187
data output pointer, 187
data transfer size, 188
lowest 32 bits of data address, 188
lowest 32 bits of device capacity, 187
name length, 187
number, 187
number of NAND flash chips, 187
out-of-band data size, 187
page size, 187
registers, 187–188
return status of controller commands, 187
top 32 bits of data address, 188
top 32 bits of device capacity, 187
NAND flash programming interface, reading/writing
blocks, 199–200
blocks with out-of-band, 198
out-of-band data, 206
page data, 206
to/from flash memory, 188
NAND flash programming interface, testing
example code, 206–210, 211–216
files, 210. See also specific files.
flash info from the Linux kernel, 206–210
NAND flash properties
description, 185
flash device layout, 185
flash info from the Linux kernel, 206–210
getting, 188
NAND_ADDR_HIGH register, 188
NAND_ADDR_LOW register, 188
NAND_CMD_BLOCK_BAD_SET command, 187
NAND_CMD_ERASE command, 187
NAND_CMD_GET_DEV_NAME CO command, 187
NAND_CMD_READ command, 187
NAND_CMD_WRITE command, 187
NAND_COMMAND register, 187
NAND_DATA register, 187
NAND_DEV register, 187
NAND_DEV_erase_SIZE register, 187
NAND_DEV.EXTRA_SIZE register, 187
NAND_DEV_FLAGS register, 187
NAND_DEV_NAME_LEN register, 187
NAND_DEV_PAGE_SIZE register, 187
NAND_DEV_SIZE_HIGH register, 187
NAND_DEV_SIZE_LOW register, 187
nand.h file, 241–243
NAND_NUM_DEV register, 187
NAND_RESULT register, 187
NAND_TRANSFER_SIZE register, 188
NAND_VERSION register, 187
Nested interrupt handler. See Interrupt handler, nested.
Newlib C library
CS3 linker scripts, 97–103
system call implementation, 104–111
Newlib C library, example code. See also specific files.
common files, 96
CS3 linker scripts, 97–104
debugging the library, 112–116
project-specific files, 96
running the library, 112–116
semihosting support, 118–122
startup code sequence, 97–103
Newlib C library, startup code sequence
   assembly initialization phase, 103
   C initialization phase, 104
   common, 97
   custom, 103–104
   hard reset phase, 103
na command, 31–32
NOR flash, booting Android from. See also NAND flash.
   introduction, 254–256
   memory relocation, 256
   RAMDISK image, creating, 256–258

objcopy command, 32
Object file formats, converting, 32
Online resources. See also Books and publications.
   CyanogenMod wiki, 332
   “Integrated Kernel Building,” 334
   “Learn about the Repo Tool...,” 355
   “Repo: Tips & Tricks,” 355
   source code for this book, 344
OOB (out-of-band) data
   definition, 185
   reading/writing, 206
   reading/writing blocks with, 198
   size, getting, 187

Period (.)
   in assembly language directives, 30
   location counter, 52, 69
   in section names, 69
Privileged modes, 66
Procedure Call Standard for the ARM Architecture, 78
Processor mode switch, discovering, 155–163
Processor modes
   changing, 150–152
   list of, 157
Program status register, nested interrupt handler, 157
Programming
   directly on hardware. See Bare metal programming,
   your first program, 29–30
   Programming Embedded System in C and C++, 5
Project-specific files, folder for, 81
PUT_CHAR register, 18

q command, 91
QEMU emulator, 11–12
qemu-system-arm command, installing, 223
Quitting debugging, 91

r command, 179
RAMDISK image
   creating, 256–258
   verifying, 273–274
ramdisk.img image file, 183–184
Read buffer command. See CMD_READ_BUFFER
   command.
   _read system call, 112
Reading/writing, NAND flash programming interface. See
   also Input/output.
   blocks, 199–200
   blocks with out-of-band, 198
   out-of-band data, 206
   page data, 206
   to/from flash memory, 188
Reading/writing to/from file, 112
Read-only data, 68
Real-time clock (RTC). See RTC (real-time clock), and
   timer.
Real-time operating system (RTOS). See also Embedded
   system programming.
   description, 5
   vs. a full operating system, 9
RealView Compilation Tools Developer Guide, 50
RealView Development Suite, 95
RealView Platform Baseboard, 5
Registers
   ALARM_HIGH, 19
   ALARM_LOW, 19
   APCS use convention, 78
   ARM register set, 67
   banked, initializing, 66–68
   BYTES_READY, 18
   CLEAR_ALARM, 19
   CLEAR_INTERRUPT, 19
   CMD, 18
   DATA_LEN, 18
   DATA_PTR, 18
   GOLDFISH_INTERRUPT_DISABLE, 127
   GOLDFISH_INTERRUPT_DISABLE_ALL, 127
   GOLDFISH_INTERRUPT_ENABLE, 127
   GOLDFISH_INTERRUPT_NUMBER, 127
   GOLDFISH_INTERRUPT_STATUS, 127
   for hardware interfaces, 17
   NAND_ADDR_HIGH, 188
   NAND_ADDR_LOW, 188
   NAND_COMMAND, 187
   NAND_DATA, 187
   NAND_DEV, 187
   NAND_DEV_ERASE_SIZE, 187
   NAND_DEV_EXTRA_SIZE, 187
   NAND_DEV_FLAGS, 187
   NAND_DEV_NAME_LEN, 187
   NAND_DEV_PAGE_SIZE, 187
   NAND_DEV_SIZE_HIGH, 187
   NAND_DEV_SIZE_LOW, 187
   NAND_NUM_DEV, 187
   NAND_RESULT, 187
   NAND_TRANSFER_SIZE, 188
   NAND_VERSION, 187
   PUT_CHAR, 18
   TIME_HIGH, 19
   TIME_LOW, 19
Serial ports, goldfish platform
  base addresses, 18
  checking the data buffer, 87–88
  debugging, console log example, 91
  getting data from, 87–89
  input/output, 88–89
  providing support for, 81–87
  sending data to, 88–89
  test cases, example code, 90–91
  unit test, 90–91
Serial ports, initializing, 112
serial_goldfish.c file
  checking data buffer, 87
  data input/output, 88–89
  goldfish serial port support, 81, 84–85
  implementing nested interrupt handler, 141
  interrupt support functions, 128
  NAND flash test program, 210–216
  Newlib C library, 96
  unit test of timer and RTC, 174
serial.h file, 240
  Setting up a development environment. See Development environment, setting up.
Sloss, Andrew N., 5
Software interrupt, triggering, 150–152
Software layers of embedded systems, 7–10
Source code for this book
  AOSP, building, 352–353
  build environment, setting up, 341–344
  CyanogenMod, building, 353–354
  organization of, 344
  virtual machine, setting up, 344
Source code for this book, building
  from the command line, 345
  from Eclipse, 346–350
Source code for this book, overview
  Part I, 345–350
  Part II, 350–352
  Part III, 352–354
Source code for this book, testing
  from the command line, 345
  from Eclipse, 346–350
Source tree, syncing, 355–356
Source-level debugging a flash image, 266–270
Sourcery CodeBench Lite, downloading, 28–29
Sourcery CodeBench trial version, downloading, 28–29
Spare area. See OOB (out-of-band) data.
SSD vs. NAND flash, 184
Stack
  banked stack pointers, initializing, 66–68
  EA (empty ascending), 66
  ED (empty descending), 66
  FA (full ascending), 66
  FD (full descending), 66
  preparing for a bare metal environment, 65–68
types of, 66
Stack pointer addresses, nested interrupt handler, 156
Stack pointers, initializing, 66–68, 72
a known configuration of U-Boot, 222–224
unit test of timer and RTC, 174

Testing, interrupt handler
alarms, 139
timer interrupts, 140

Testing, nested interrupt handler
nested interrupt handler, 155–163
RTC (real-time clock) unit test, 174–179
setting breakpoints, 158–163
software interrupts, 163–164
system calls, 163–164
timer, 174–179
timer interrupt, 140

Timer
description, 172–173
timer_init() function, 172
U-Boot API, 172

Timer, setting/increasing/resetting, 179
timer interrupts, 178–179

Toolchains
cross-compilation, identifying, 31
downloading, 26–29
installing, 26–29
uclibc, library variant, 95
__udelay() function, 172
udelay_masked() function, 172
umount command, 274
Underscore (_),
in assembly language labels, 30
in symbol names, 69
Unit test
Newlib C library, 112–116
serial ports, 90–91
Uppercase letters in constant names, 69
userdata.img, 257
userdata.img image file, 183
VERBOSE option, 37–38
Verifying
Android kernel, 273–274
RAMDISK image, 273–274
Versatile PB vs. goldfish, 229
versatileqemu program, 222–224
Version information, 80
Virtual devices
configuring, 14–16
setting up, 284–287
Virtual hardware
overview, 13–14
vs. real hardware, 11
VirtualBox, 344
Virtualization environment. See also Development environment.
definition, 6
learning embedded system programming, 6
VMA (virtual memory address), 57
VMware Player, 344
void goldfish_clear_timer() function, 172
void goldfish_set_timer() function, 172
Wildcard character, asterisk (*), 52
word directive, 43–44
Wright, Chris, 5
Write buffer command. See CMD_WRITE_BUFFER command._write system call, 112
x command, 179
yaffs_ubuntu_glue.c file, 243
Yaghmour, Karim, 5, 249, 288
ydevconfig command, 272
ydevsys command, 272
ymount command, 273
yrzm command, 273