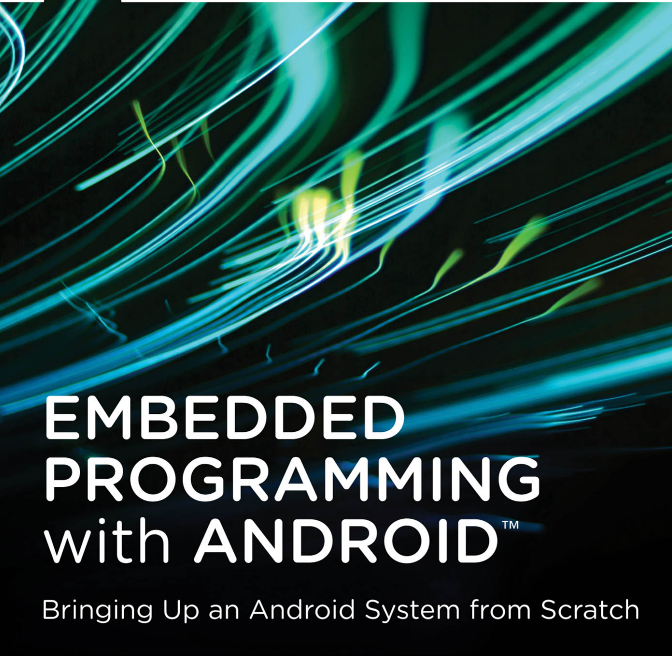
Zigurd MEDNIEKS
Series Editor

ANDROID™
DEEP
DIVE

Roger YE



EMBEDDED PROGRAMMING with ANDROID™

Bringing Up an Android System from Scratch

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Embedded Programming with Android™

About the Android Deep Dive Series

Zigurd Mednieks, Series Editor

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

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact international@pearsoned.com.

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data
Ye, Roger, author.
Embedded programming with Android : bringing up an Android system from scratch / Roger Ye.
pages cm
Includes index.
ISBN 978-0-13-403000-5 (pbk. : alk. paper)—ISBN 0-13-403000-1 (pbk. : alk. paper)
1. Android (Electronic resource) 2. Embedded computer systems—Programming.
3. Application software—Development. 4. Emulators (Computer programs)
5. Smartphones—Programming. I. Title.
QA76.76.A65Y438 2016
004.167—dc23
2015022900

ISBN-13: 978-0-13-403000-5
ISBN-10: 0-13-403000-1

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, August 2015

Editor-in-Chief
Mark L. Taub

Executive Editor
Laura Lewin

Development Editor
Michael Thurston

Managing Editor
John Fuller

Project Editor
Elizabeth Ryan

Copy Editor
Jill Hobbs

Indexer
Infodex Indexing Services

Proofreader
Linda Begley

Technical Reviewers
Zigurd Mednieks
Blake Meike

Editorial Assistant
Olivia Basegio

Cover Designer
Chuti Prasertsith

Compositor
codeMantra US, LLC

❖

To the programmers who have great interest in embedded systems
and the latest computing devices

❖

This page intentionally left blank

Contents in Brief

This page intentionally left blank

Contents

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:
>
> *http://git-scm.com/book/en/v2*
>
> 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.

```
#
# 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.
#
ifneq (${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.

Acknowledgments

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.

This page intentionally left blank

About the Author

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

This page intentionally left blank

10
Using U-Boot to Boot the Goldfish Kernel

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:

```
$ 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:

```
$ 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:

```
$ 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`.

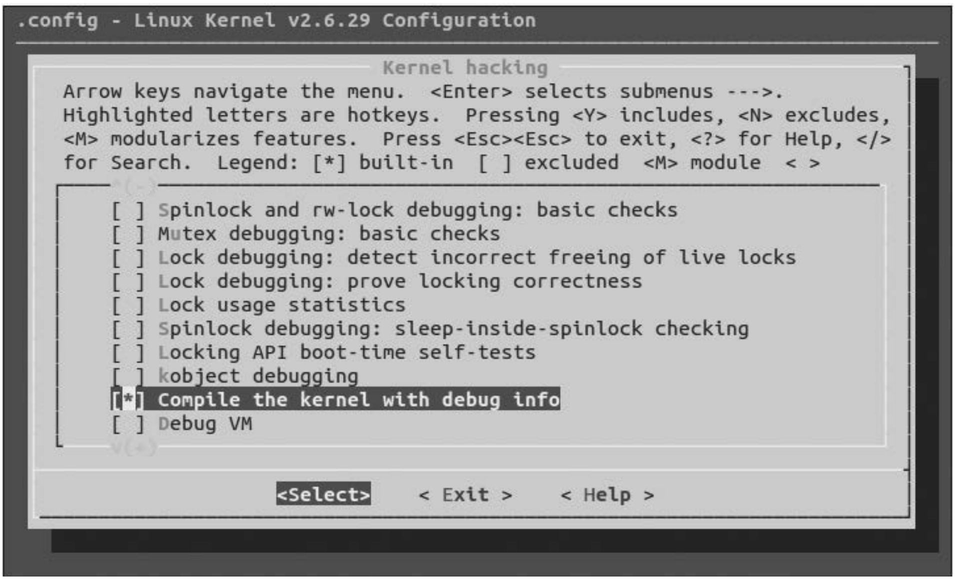


Figure 10.1 Enabling debugging in `menuconfig`

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 gdbserver 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
```

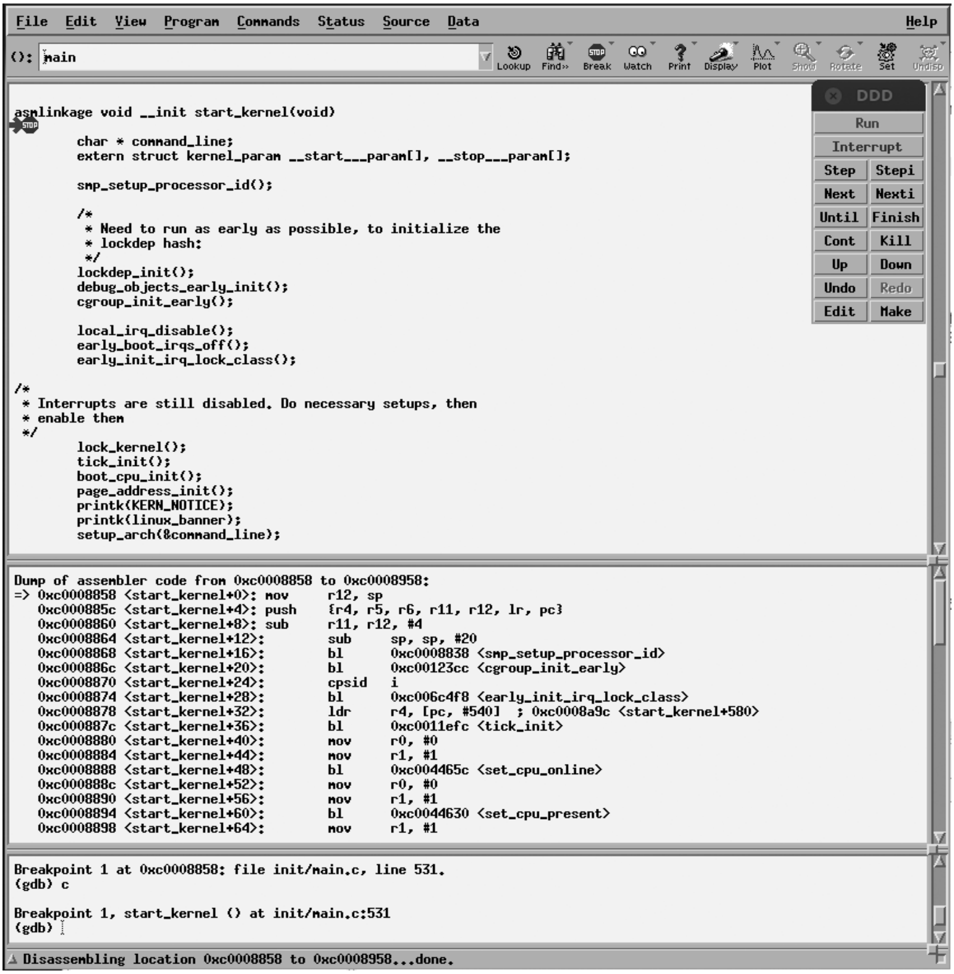


Figure 10.2 Boot-up stop at `start_kernel`

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.

```
<6>yaffs: dev is 32505858 name is "mtdblock2"
yaffs: dev is 32505858 name is "mtdblock2"
<6>yaffs: passed flags ""
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_recover
y'
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 13:53:51.1612
96804 UTC)
request_suspend_state: wakeup (3->0) at 35350000414 (2013-05-25 13:53:51.1612968
04 UTC)

shell@android:/ $
```

Figure 10.3 Linux console after boot-up

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

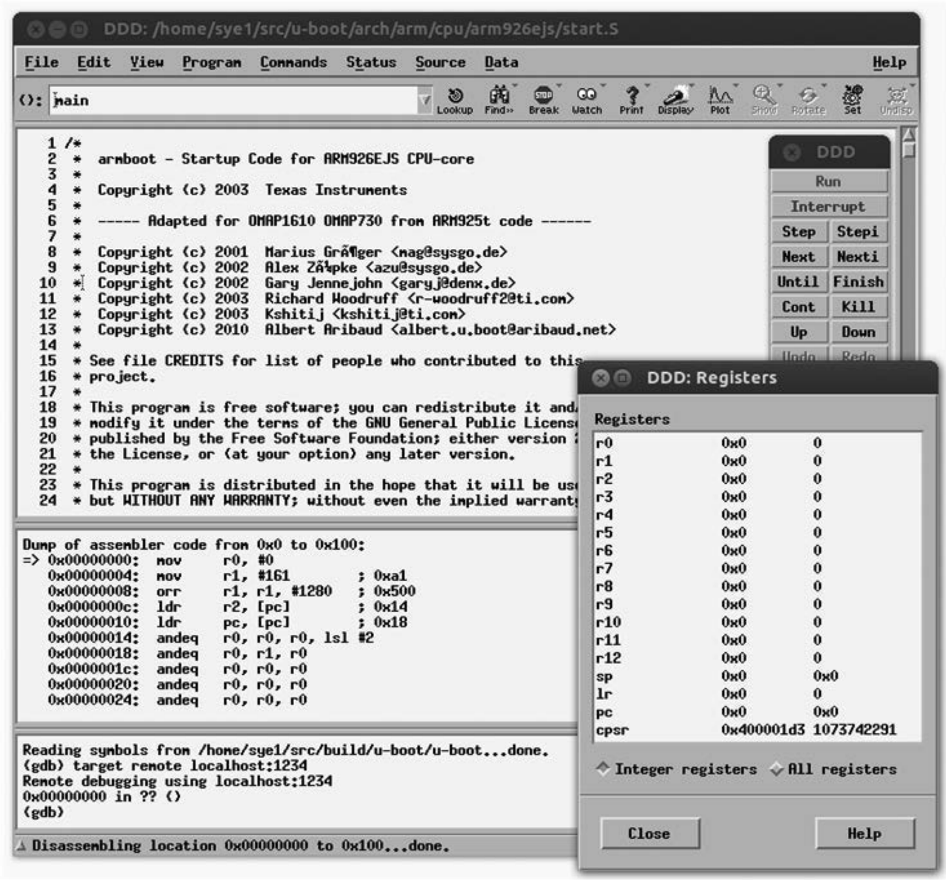


Figure 10.4 Memory dump of mini-bootloader at reset

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 0x1ff59000 (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

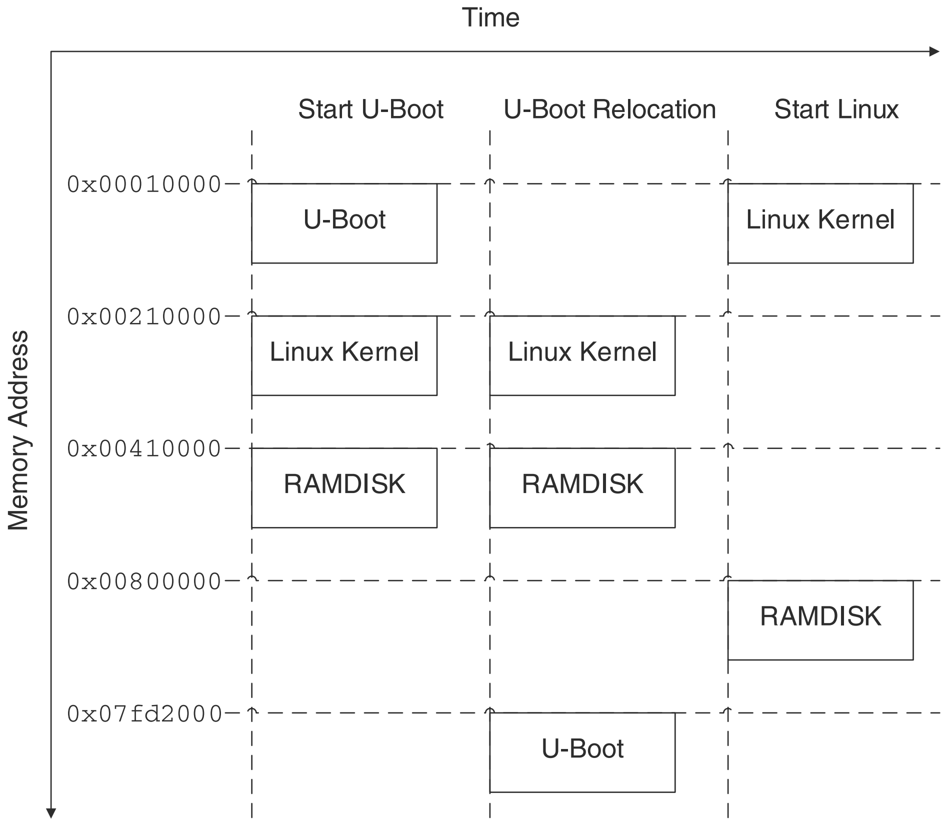


Figure 10.5 Memory relocation during boot-up

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 process by changing the RAMDISK content. 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 an 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:

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

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

```
$ 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:

```
shell@android:/ $ mount
rootfs / rootfs ro 0 0
tmpfs /dev tmpfs rw,nosuid,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:

```
$ 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:

```
$ 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:

```
$ 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:

```
$ 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.-rc1-00003-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: 1fff0000

Top of RAM usable for U-Boot at: 1fff0000

Reserving 116k for U-Boot at: 1ffd2000

Reserving 136k for malloc() at: 1ffb0000

Reserving 32 Bytes for Board Info at: 1ffaffe0

Reserving 120 Bytes for Global Data at: 1ffaff68

Reserving 8192 Bytes for IRQ stack at: 1ffadf68

New Stack Pointer is: 1ffadf58

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: 1ffd2000

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"
```

```
INSERT: table 1ffeb724, filled 5/89 rv 1ffb0154 ==> name="bootfile" value="/
tftpboot/uImage"
INSERT: free(data = 1ffb0008)
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
Goldfish # iminfo 0x210000

## Checking Image at 00210000 ...
   Legacy image found
   Image Name:
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    1722596 Bytes = 1.6 MiB
   Load Address: 00010000
   Entry Point:  00010000
   Verifying Checksum ... OK
Goldfish # iminfo 0x410000

## Checking Image at 00410000 ...
   Legacy image found
   Image Name:
   Image Type:   ARM Linux RAMDisk Image (uncompressed)
   Data Size:    187687 Bytes = 183.3 KiB
   Load Address: 00800000
   Entry Point:  00800000
   Verifying Checksum ... OK
Goldfish #
```

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 0x1ffadb10 *  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
```

```
  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 ff003000 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: SMC91C11xFD (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
device-mapper: ioctl: 4.14.0-ioctl (2008-04-23) initialised: dm-devel@redhat.com
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 2014-02-20 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"
```

```
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: untracked pid 39 exited
eth0: link up
shell@android:/ $ warning: 'zygote' uses 32-bit capabilities (legacy support in
use)
```

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`.

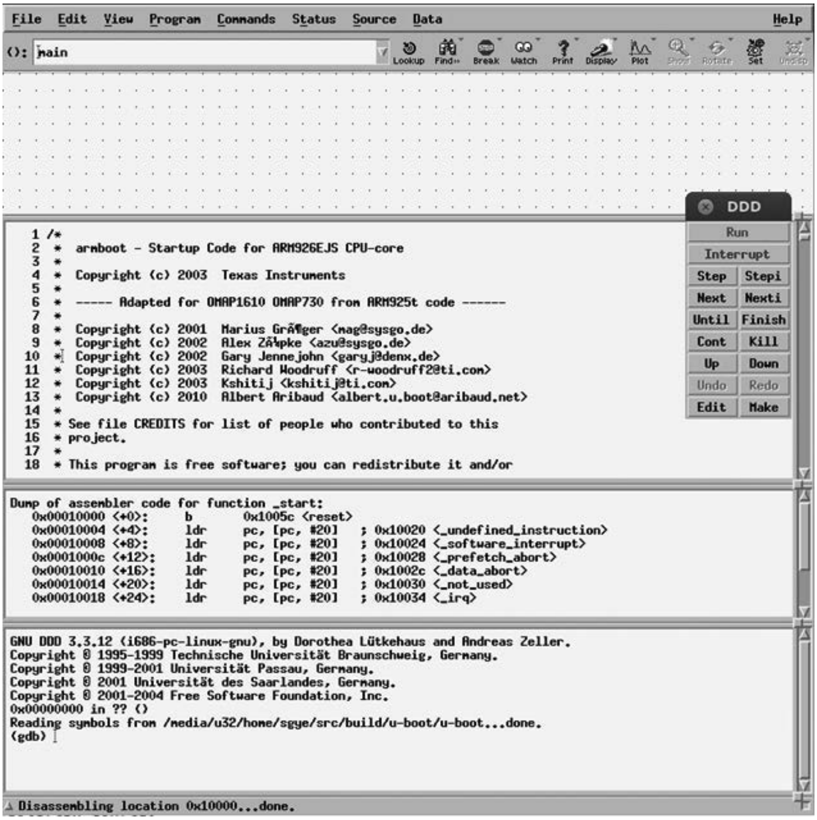


Figure 10.6 Loading U-Boot to gdb

Example 10.1 GDB Startup Script for U-Boot (u-boot.gdb)

```
# 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`.

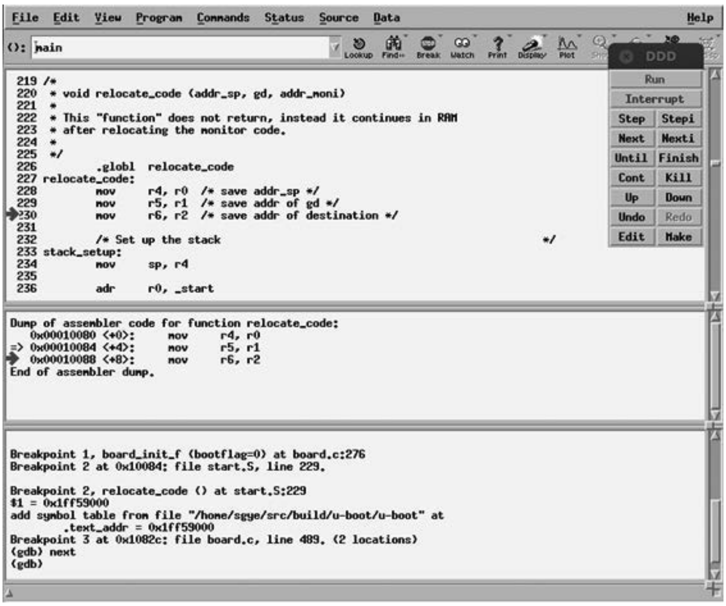


Figure 10.7 Reload the U-Boot symbol after relocation

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:

```
(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.

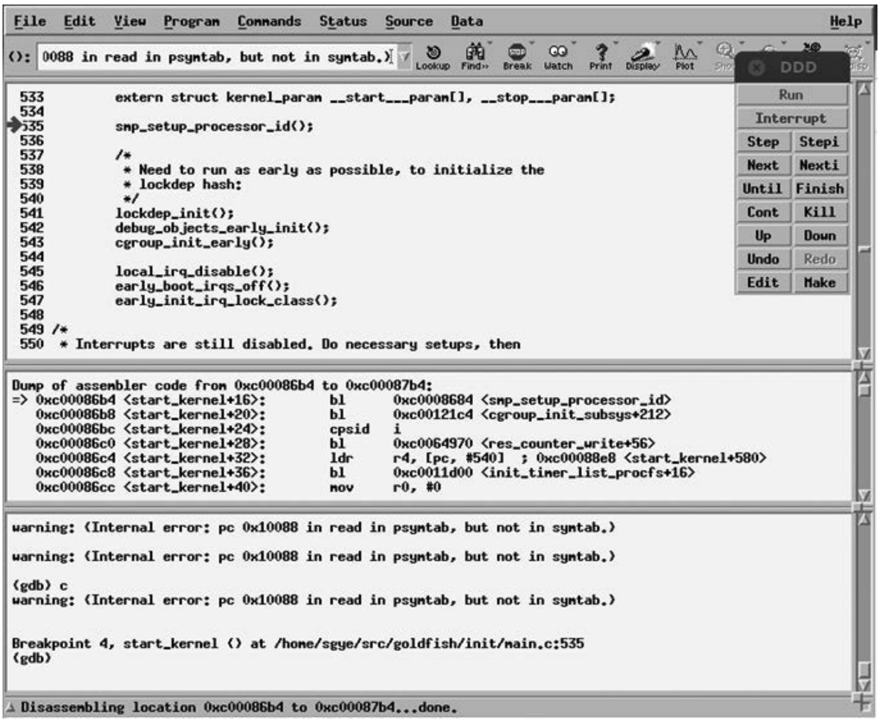


Figure 10.8 The goldfish kernel at `start_kernel()`

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:

```
$ 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: 1fff0000

Top of RAM usable for U-Boot at: 1fff0000

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(), gtty.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
```

```
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 androidboot.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 1fb572f8 ==> 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:
   Image Type:   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
```

```
Copy sys/zImage.uimg to 0x00210000...     [DONE]
Goldfish # iminfo 0x210000

## Checking Image at 00210000 ...
   Legacy image found
   Image Name:
   Image Type:   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 0x210000 0x410000
## Current stack ends at 0x1fb54b10 *  kernel: cmdline image address = 0x00210000
## Booting kernel from Legacy Image at 00210000 ...
   Image Name:
   Image Type:   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:
   Image Type:   ARM Linux RAMDisk Image (uncompressed)
   Data Size:    187703 Bytes = 183.3 KiB
   Load Address: 00800000
   Entry Point:  00800000
   ramdisk start = 0x00800000, ramdisk end = 0x0082dd37
   Loading Kernel Image ... OK
CACHE: Misaligned operation at range [00010000, 006a2790]
```

```
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)
```

```
Calibrating delay loop... 452.19 BogoMIPS (lpj=2260992)
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 ff003000 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 Mar  7 2014 15:57:44 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: 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
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
device-mapper: ioctl: 4.14.0-ioctl (2008-04-23) initialised: dm-devel@redhat.com
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

B

G