Bare-metal programming for ARM

A hands-on guide

Daniels Umanovskis

Contents

0 Introduction

Modern programming takes many forms. There's web development, desktop application development, mobile development and more. Embedded programming is one of the areas of programming, and can be radically different from the others. Embedded programming means programming for a computer that's mainly intended to be embedded within a larger system, and the embedded computer is usually responsible for a particular task, instead of being a general-purpose computing device. The system where it's embedded might be a simple pocket calculator, or an industrial robot, or a spaceship. Some embedded devices are microcontrollers with very little memory and low frequencies, others are more powerful.

Embedded computers may be running a fully-fleged operating system, or a minimalistic system that just provides some scheduling of real-time functions. In cases when there's no operating system at all, the computer is said to be *bare metal*, and consequently *bare metal programming* is programming directly for a (micro-)computer that lacks an operating system. Bare metal programming can be both highly challenging and very different from other types of programming. Code interfaces directly with the underlying hardware, and common abstractions aren't available. There are no files, processes or command lines. You cannot even get the simplest C code to work without some preparatory steps. And, in one of the biggest challenges, failures tend to be absolute and mysterious. It's not uncommon to see embedded developers break out tools such as voltmeters and oscilloscopes to debug their software.

Modern embedded hardware comes in very many types, but the field is dominated by CPUs implementing an ARM architecture. Smartphones and other mobile devices often run Qualcomm Snapdragon or Apple A-series CPUs, which are all based on the ARM architecture. Among microcontrollers, ARM Cortex-M and Cortex-R series CPU cores are very popular. The ARM architectures play a very significant role in modern computing.

The subject of this ebook is bare-metal programming in C for an ARM system. Specifically, the ARMv7-A architecture is used, which is the last purely 32-bit ARM architecture, unlike the newer ARMv8/AArch64. The -A suffix in ARMv7-A indicates the A profile, which is intended for more resource-intensive applications. The corresponding microcontroller architecture is ARMv7-M.

Note that this is not a tutorial on how to write an OS. Some of the topics covered in this ebook are relevant for OS development, but there are many OS-specific aspects that are not covered here.

Target audience

This ebook is aimed at people who have an interest in low-level programming, and in seeing how to build a system from the ground up. Topics covered include system startup, driver development and low-level memory management. For the most part, the chapters cover things from a practical perspective, by building something, although background theory is provided.

The reader should be familiar with C programming. This is not a C tutorial, and even though there are occasional notes on the language, the ebook is probably difficult to follow without having programmed in C. Some minimal exposure to an assembly language and understanding of computer architecture are very useful, though the assembly code presented is explained line by line as it's introduced.

It's also helpful to be familiar with Linux on a basic level. You should be able to navigate directories, run shell scripts and do basic troubleshooting if something doesn't work - fortunately, even for an inexperienced Linux user, a simple online search is often enough to solve a problem. The ebook assumes all development is done on Linux, although it should be possible to do it on OS X and even on Windows with some creativity.

Experienced embedded developers are unlikely to find much value in this text.

Formatting and terminology

The ebook tries to for the most part follow usual conventions for a programming-related text. Commands, bits of code or references to variables are `formatted like code`, with bigger code snippets presented separately like this:

```
1 void do_amazing_things(void) {
2     int answer = 42;
3     /* A lot happens here! */
4 }
```

If you are reading the PDF version, note that longer lines of code have to get wrapped to fit within the page, but the indentation and line numbers inside each code block should help keep things clear.

Due to some unfortunate historical legacy, there are two different definitions for data sizes in common use. There's the binary definition, where a kilobyte is 1024 bytes, and the metric definition, where a kilobyte is 1000 bytes. Throughout this ebook, all references to data quantities are in the binary sense. The meaning of "billion" in the book is `10^9`.

Source code

For each chapter, the corresponding source code is available. If you're reading this on GitHub, you can explore the repository and check its readme file for more information. If you're reading the PDF version or other standalone copy, you can head to the GitHub repository to access the source code, and perhaps updates to this ebook. The repository URL is `https://github.com/umanovskis/baremetal-arm/`.

Licensing

Credits and acknowledgments

The PDF version of this ebook is typeset using the Eisvogel LaTeX template by Pascal Wagler.

1 Environment setup

In this chapter, I'll cover the basic environment setup to get started with ARM bare-metal programming using an emulator. Some familiarity with using Linux is assumed. You don't need to be a Linux expert, but you should be able to use the command line and do some basic troubleshooting if the system doesn't behave as expected.

Linux

The first prerequisite is getting a Linux system up and running. Hopefully you are already familiar with Linux and have some kind of Linux running. Otherwise you should install Linux, or set up a virtual machine running Linux.

If you are running Windows and want to run a virtual Linux, VirtualBox is recommended. As for Linux distributions, any modern distribution should be fine, although in some cases you might need to install software manually. I use Linux Mint Debian Edition, and double-check most of the work in a virtual machine running Ubuntu, which is the most popular Linux distribution for beginners.

QEMU

To emulate an ARM machine, we will be using QEMU, a powerful emulation and virtualization tool that works with a variety of architectures. While the code we write should eventually be able to boot on a real ARM device, it is much easier to start with an emulator. Why?

- No additional hardware is needed.
- You don't have to worry about software flashing / download process.
- You have much better tools to inspect the state of the emulated hardware. When working with real hardware, you would need a few drivers to get meaningful information from the software, or use other more difficult methods.

Since QEMU supports a wide range of systems, we'll need to install the ARM version. On Debian/Ubuntu based systems, the `qemu-system-arm` package will provide what you need, so let's just go ahead and install it:

```
1 sudo apt-get install qemu-system-arm
```

GCC cross-compiler toolchain

The next step is the installation of a cross-compiler toolchain. You cannot use the regular `gcc` compiler to build code that will run on an ARM system, instead you'll need a cross-compiler. What is a cross-compiler exactly? It's simply a compiler that runs on one platform but creates executables for another platform. In our case, we're running Linux on the x86-64 platform, and we want executables for ARM, so a cross compiler is the solution to that.

The GNU build tools, and by extension GCC, use the concept of a *target triplet* to describe a platform. The triplet lists the platform's architecture, vendor and operating system or binary interface type. The vendor part of target triplets is generally irrelevant. You can look up your own machine's target triplet by running `gcc -dumpmachine`. I get `x86_64-linux-gnu`, yours will likely be the same or similar.

To compile for ARM, we need to select the correct cross-compiler toolchain, that is, the toolchain with a target triplet matching our actual target. The fairly widespread `gcc-arm-linux-gnueabi` toolchain will **not** work for our needs, and you can probably guess why – the name indicates that the toolchain is intended to compile code for ARM devices running Linux. We're going to do bare-metal programming, so no Linux running on the target system.

The toolchain we need is `gcc-arm-none-eabi`. We will need a version with GCC 6 or newer, for when we later use U-Boot. On Ubuntu, you should be able to simply install the toolchain:

```
1 sudo apt-get install gcc-arm-none-eabi
```

You can run `arm-none-eabi-gcc --version` to check the version number. If you're using a distribution that offers an old version of the package, you can download the toolchain from ARM directly. In that case, it's recommended that you add the toolchain's folder to your environment's `PATH` after extracting it somewhere.

Build system essentials

Finally, we need the essential components of a build system. In the coming examples, we'll be using the standard Make build tool, as well as CMake. Debian-based systems provide a handy package called `build-essential`, which installs Make and other relevant programs. CMake is available in a package called `cmake`, so the installation is simple:

```
1 sudo apt-get install build-essential cmake
```

On some Linux variants, you might also need to install `bison` and `flex` if they are not already present. Those tools are also required to build U-Boot.

```
1 sudo apt-get install bison flex
```

`sort -R ~/facts-and-trivia | head -n1`

The `flex` program is an implementation of `lex`, a standard lexical analyzer first developed in the mid-1970s by Mike Lesk and Eric Schmidt, who served as the chairman of Google for some years.

With this, your system should now have everything that is necessary to compile programs for ARM and run them in an emulated machine. In the next chapter, we'll continue our introduction by booting the emulated machine and giving some of the just-installed tools a spin.

2 The first boot

We'll continue our introduction to bare-metal ARM by starting an emulated ARM machine in QEMU, and using the cross-compiler toolchain to load the simplest possible code into it.

Let us run QEMU for the very first time, with the following command:

```
1 qemu-system-arm -M vexpress-a9 -m 32M -no-reboot -nographic -monitor
     telnet:127.0.0.1:1234,server,nowait
```

The QEMU machine will spin up, briefly seem to do nothing, and then crash with an error message. The crash is to be expected - we did not provide any executable to run so of course our emulated system cannot accomplish anything. For documenation of the QEMU command line, you can check `man qemu-doc` and online, but let's go through the command we used and break it down into parts.

- `-M vexpress-a9`. The `-M` switch selects the specific machine to be emulated. The ARM Versatile Express is an ARM platform intended for prototyping and testing. Hardware boards with the Versatile Express platform exist, and the platform is also a common choice for testing with emulation. The `vexpress-a9` variant has the Cortex A9 CPU, which powers a wide variety of embedded devices that perform computationally-intensive tasks.
- `-m 32M`. This simply sets the RAM of the emulated machine to 32 megabytes.
- `-no-reboot`. Don't reboot the system if it crashes.
- `-nographic`. Run QEMU as a command-line application, outputting everything to a terminal console. The serial port is also redirected to the terminal.
- `-monitor telnet:127.0.0.1:1234,server,nowait`. One of the advantages of QEMU is that it comes with a powerful *QEMU monitor*, an interface to examine the emulated machine and control it. Here we say that the monitor should run on localhost, port `1234`, with the `server`, `nowait` options meaning that QEMU will provide a telnet server but will continue running even if nobody connects to it.

That's it for the first step - now you have a command to start an ARM system, although it will not do anything except crashing with a message like `qemu-system-arm: Trying to execute code outside RAM or ROM at 0x04000000`.

The first hang

Writing some code

Now we want to write some code, turn it into an executable that can run on an ARM system, and run it in QEMU. This will also serve as our first cross-compilation attempt, so let's go with the simplest code possible - we will write a value to one of the CPU registers, and then enter an infinite loop, letting our (emulated) hardware hang indefinitely. Since we're doing bare-metal programming and have no form of runtime or operating system, we have to write the code in assembly language. Create a file called `startup.s` and write the following code:

```
1 ldr r2,str1
2 b .
3 str1: .word 0xDEADBEEF
```

Line by line:

1. We load the value at label `str1` (which we will define shortly) into the register `R2`, which is one of the general-purpose registers in the ARM architecture.
2. We enter an infinite loop. The period `.` is short-hand for *current address*, so `b .` means "branch to the current instruction address", which is an infinite loop.
3. We allocate a word (4 bytes) with the value `0xDEADBEEF`, and give it the label `str1`. The value `0xDEADBEEF` is a distinctive value that we should easily notice. Writing such values is a common trick in low-level debugging, and `0xDEADBEEF` is often used to indicate free memory or a general software crash. Why will this work if we're in an infinite loop? Because this is not executable code, it's just an instruction to the assembler to allocate the 4-byte word here.

```
sort -R ~/facts-and-trivia | head -n1
```

Memorable hexadecimal values like `0xDEADBEEF` have a long tradition, with different vendors and systems having their own constants. Wikipedia has a separate article on these magic values.

Assembling it

Next we need to compile the code. Since we only wrote assembly code, compilation is not actually relevant, so we will just assemble and link the code. The first time, let's do this manually to see how to use the cross-compiler toolchain correctly. What we're doing here is very much not optimal even for an example as simple as this, and we'll improve the state of things soon.

First, we need to assemble `startup.s`, which we can do like this, telling the GNU Assembler (`as`) to place the output in `startup.o`.

```
1 arm-none-eabi-as -o startup.o startup.s
```

We do not yet have any C files, so we can go ahead and link the object file, obtaining an executable.

```
1 arm-none-eabi-ld -o first-hang.elf startup.o
```

This will create the executable file `first-hang.elf`. You will also see a warning about missing `_start`. The linker expects your code to include a `_start` symbol, which is normally where the execution would start from. We can ignore this now because we only need the ELF file as an intermediate step anyway. The `first-hang.elf` which you obtained is a sizable executable, reaching 33 kilobytes on my system. ELF executables are the standard for Linux and other Unix-like systems, but they need to be loaded and executed by an operating system, which we do not have. An ELF executable is therefore not something we can use on bare metal, so the next step is to convert the ELF to a raw binary dump of the code, like this:

```
1 arm-none-eabi-objcopy -O binary first-hang.elf first-hang.bin
```

The resulting `first-hang.bin` is just a 12-byte file, that's all the space necessary for the code we wrote in `startup.s`. If we look at the hexdump of this file, we'll see `00000000 00 20 9f e5 fe ff ff ea ef be ad de`. You can recognize our `0xDEADBEEF` constant at the end. The first eight bytes are our assembly instructions in raw form. The code starts with `0020 9fe5`. The `ldr` instruction has the opcode `e5`, then `9f` is a reference to the program counter (PC) register, and the `20` refers to `R2`, meaning this is in fact `ldr r2, [pc]` encoded.

Looking at the hexdump of a binary and trying to match the bytes to assembly instructions is uncommon even for low-level programming. It is somewhat useful here as an illustration, to see how we can go from writing `startup.s` to code executable by an ARM CPU, but this is more detail than you would typically need.

And... Blastoff!

We can finally run our code on the ARM machine! Let's do so.

```
qemu-system-arm -M vexpress-a9 -m 32M -no-reboot -nographic -monitor
    telnet:127.0.0.1:1234,server,nowait -kernel first-hang.bin
```

This runs QEMU like previously, but we also pass `-kernel first-hang.bin`, indicating that we want to load our binary file into the emulated machine. This time you should see QEMU hang indefinitely. The QEMU monitor allows us to read the emulated machine's registers, among other things, so we can check whether our code has actually executed. Open a telnet connection in another terminal with `telnet localhost 1234`, which should drop you into the QEMU monitor's command line, looking something like this:

```
QEMU 2.8.1 monitor - type 'help' for more information
(qemu)
```

At the `(qemu)` prompt, type `info registers`. That's the monitor command to view registers. Near the beginning of the output, you should spot our `0xDEADBEEF` constant that has been loaded into R2:

```
R00=00000000 R01=000008e0 R02=deadbeef R03=00000000
```

This means that yes indeed, QEMU has successfully executed the code we wrote. Not at all fancy, but it worked. We have our first register write and hang.

What we did wrong

Our code worked, but even in this small example we didn't really do things the right way.

Memory mappings

One issue is that we didn't explicitly specify any start symbol that would show where our program should begin executing. It works because when the CPU starts up, it begins executing from address `0x0`, and we have placed a valid instruction at that address. But it could easily go wrong. Consider this variation of `startup.s`, where we move the third line to the beginning.

```
str1: .word 0xDEADBEEF
ldr r2,str1
b .
```

That is still valid assembly and feels like it should work, but it wouldn't - the constant `0xDEADBEEF` would end up at address `0x0`, and that's not a valid instruction to begin the program with. Moreover, even starting at address `0x0` isn't really correct. On a real system, the interrupt vector table should be located at address `0x0`, and the general boot process should first have the bootloader starting, and after that switch execution to your code, which is loaded somewhere else in memory.

QEMU is primarily used for running Linux or other Unix-like kernels, which is reflected in how it's normally started. When we start QEMU with `-kernel first-hang.bin`, QEMU acts as if booting such a kernel. It copies our code to the memory location `0x10000`, that is, a 64 kilobyte offset from the beginning of RAM. Then it starts executing from the address `0x0`, where QEMU already has some startup code meant to prepare the machine and jump to the kernel.

Sounds like we should be able to find our `first-hang.bin` at `0x10000` in the QEMU memory then. Let's try do to that in the QEMU monitor, using the `xp` command which displays physical memory. In the QEMU monitor prompt, type `xp /4w 0x100000` to display the four words starting with that memory address.

```
0000000000100000: 0x00000000 0x00000000 0x00000000 0x00000000
```

Everything is zero! If you check the address `0x0`, you will find the same. How come?

The answer is memory mapping - the address space of the device encompasses more than just the RAM. It's time to consult the most important document when developing for a particular device, its *Technical Reference Manual*, or TRM for short. The TRM for any embedded device is likely to have a section called "memory map" or something along those lines. The TRM for our device is available from ARM, and it indeed contains a memory map. (Note: when working with any device, downloading a PDF version of the TRM is a very good idea.) In this memory map, we can see that the device's RAM (denoted as "local DDR2") begins at `0x60000000`.

That means we have to add `0x60000000` to RAM addresses to obtain the physical address, so our `0x10000` where we expect the binary code to be loaded is at physical address `0x60010000`. Let's check if we can find the code at that address: `xp /4w 0x60010000` shows us:

```
0000000060010000: 0xe59f2000 0xeaffffe 0xdeadbeef 0x00000000
```

There it is indeed, our `first-hang.bin` loaded into memory!

Creating the vector table

Having our code start at address `0x0` isn't acceptable as explained before, as that is the address where the interrupt vector table is expected. We should also not rely on things just working out, with help from QEMU or without, and should explicitly specify the entry point for our program. Finally, we should separate code and data, placing them in separate sections. Let's start by improving our `startup.s` a bit:

```
.section .vector_table, "x"
.global _Reset
_Reset:
    b Reset_Handler
    b . /* 0x4  Undefined Instruction */
    b . /* 0x8  Software Interrupt */
    b . /* 0xC  Prefetch Abort */
    b . /* 0x10 Data Abort */
    b . /* 0x14 Reserved */
    b . /* 0x18 IRQ */
    b . /* 0x1C FIQ */

.section .text
Reset_Handler:
    ldr r2, str1
    b .
    str1: .word 0xDEADBEEF
```

Here are the things we're doing differently this time:

1. We're creating the vector table at address `0x0`, putting it in a separate section called `.vector_table`, and declaring the global symbol `_Reset` to point to its beginning. We leave most items in the vector table undefined, except for the reset vector, where we place the instruction `b Reset_Handler`.
2. We moved our executable code to the `.text` section, which is the standard section for code. The `Reset_Handler` label points to the code so that the reset interrupt vector will jump to it.

Creating the linker script

Linker scripts are a key component in building embedded software. They provide the linker with information on how to place the various sections in memory, among other things. Let's create a linker script for our simple program, call it `linkscript.ld`.

```
ENTRY(_Reset)

SECTIONS
{
    . = 0x0;
    .text : { startup.o (.vector_table) *(.text) }
    . = ALIGN(8);
}
```

This script tells the linker that the program's entry point is at the global symbol `_Entry`, which we export from `startup.s`. Then the script goes on to list the section layout. Starting at address `0x0`, we create the `.text` section for code, consisting first of the `.vector_table` section from `startup.o`, and then any and all other `.text` sections. We align the code to an 8-byte boundary as well.

What's a linker anyway?

Indeed, what's a linker and why are we using one? As developers, we often say "compilation" when referring to the process by which source code turns into an executable. More accurately though, compilation is just one step, normally followed by linking, and it's this compile-and-link process that often gets simply called compilation. The confusion isn't helped by the fact that compiling and linking are usually invoked with the same command in most tools. Whether you're using an IDE or using GCC from the command line, compilation and linking will usually occur together.

The compiler takes source code and produces *object files* as the output, these files usually have the `.o` extension. A linker takes one or more object files, possibly adds external libraries, and links it all into an executable. In any non-trivial program, each object file is likely to refer to functions that are contained in other object files, and resolving those dependencies is part of the linker's job. So linkers themselves are nothing specific to embedded programming, but due to the low abstraction level available when programming for bare metal, it's common to require more control over the linker's actions.

Linker scripts like the one above, in the broadest terms, tell the linker how to do its job. For now we're just giving it simple instructions, but later we'll write a more sophisticated linker script.

Hanging again - but better

We can now build the updated software. Let's do that in a similar manner to before:

```
arm-none-eabi-as -o startup.o startup.s
arm-none-eabi-ld -T linkscript.ld -o better-hang.elf startup.o
arm-none-eabi-objcopy -O binary better-hang.elf better-hang.bin
```

Note the addition of `-T linkscript.ld` to the linker command, specifying to use our newly created linker script. We still cannot use the ELF file directly, but we could use `objdump` to verify that our linkscript changed things. Call `arm-none-eabi-objdump -h better-hang.elf` to see the list of sections. You'll notice the `.text` section. And if you use `objdump` to view `startup.o`, you'll also see `.vector_table`. You can even observe that the sizes of `.vector_table` and `.text` in `startup.o` add up to the size of `.text` in the ELF file, further indicating that things are probably as we wanted.

We can now once again run the software in QEMU with `qemu-system-arm -M vexpress-a9 -m 32M -no-reboot -nographic -monitor telnet:127.0.0.1:1234,server,nowait -kernel better-hang.bin` and observe the same results as before, and happily knowing things are now done in a more proper way.

In the next chapter, we will continue by introducing a bootloader into our experiments.

3 Adding a bootloader

Introduction

The *bootloader* is a critical piece of software that is necessary to get the hardware into a usable state, and load other more useful programs, such as an operating system. On PCs and other fully-featured devices, common bootloaders include GNU GRUB (which is most likely used to boot your Linux system), `bootmgr` (for modern versions of MS Windows) and others. Developing bootloaders is a separate and complicated subject. Bootloaders are generally full of esoteric, highly architecture-specific code, and in my opinion learning about bootloaders if fun, but the knowledge is also somewhat less transferable to other areas of development.

Writing your own bootloader is certainly something that could be attempted, but in this series of posts we will continue by doing something that is usually done in embedded development, namely using Das U-Boot.

Das U-Boot, usually referred to just as U-Boot, is a very popular bootloader for embedded devices. It supports a number of architectures, including ARM, and has pre-made configurations for a very large number of devices, including the Versatile Express series that we're using. Our goal in this article will be to build U-Boot, and combine it with our previously built software. This will not, strictly speaking, change anything significant while we are running on an emulated target in QEMU, but we would need a bootloader in order to run on real hardware.

We will, in this article, change our boot sequence so that U-Boot starts, and then finds our program on a simulated SD card, and subsequently boots it. I will only provide basic explanations for some of the steps, because we'll mostly be dealing with QEMU and Linux specifics here, not really related to ARM programming.

Preparing U-Boot

First, you should download U-Boot. You could clone the project's source tree, but the easiest way is to download a release from the official FTP server. For writing this, I used `u-boot-2018.09`. This is also

the reason why your cross-compiler toolchain needs `gcc` of at least version 6 - earlier versions cannot compile U-Boot.

After downloading U-Boot and extracting the sources (or cloning them), you need to run two commands in the U-Boot folder.

```
make vexpress_ca9x4_config ARCH=arm CROSS_COMPILE=arm-none-eabi-
```

This command will prepare some U-Boot configuration, indicating that we want it for the ARM architecture and, more specifically, we want to use the `vexpress_ca9x4` configuration, which corresponds to the `CoreTile Express A9x4` implementation of the Versatile Express platform that we're using.' The configuration command should only take a few seconds to run, after which we can build U-Boot:

```
make all ARCH=arm CROSS_COMPILE=arm-none-eabi-
```

If everything goes well, you should, after a short build process, see the file `u-boot` and `u-boot.bin` created. You can quickly test by running QEMU as usual, except you start U-Boot, providing `-kernel u-boot` on the command line (note that you're booting `u-boot` and not `u-boot.bin`). You should see U-Boot output some information, and you can drop into the U-Boot command mode if you hit a key when prompted.

Having confirmed that you can run U-Boot, make a couple of small modifications to it. In `configs/vexpress_ca9x4_defconfig`, change the `CONFIG_BOOTCOMMAND` line to the following:

```
CONFIG_BOOTCOMMAND="run mmc_elf_bootcmd"
```

The purpose of that will become clear a bit later on. Then open `include/config_distro_bootcmd.h` and go to the end of the file. Find the last line that says `done\0` and edit from there so that the file looks like this:

```
        "done\0"                                                \
    \
    "bootcmd_bare_arm="                                         \
        "mmc dev 0;"                                            \
        "ext2load mmc 0 0x60000000 bare-arm.uimg;"              \
        "bootm 0x60000000;"                                     \
        "\0"
```

Note that in the above snippet, the first line with `done\0` was already in the file, but we add a backslash `\` to the end, and then we add the subsequent lines. See the edited file in the repository. Regenerate the U-Boot config and rebuild it:

```
make vexpress_ca9x4_config ARCH=arm CROSS_COMPILE=arm-none-eabi-
make all ARCH=arm CROSS_COMPILE=arm-none-eabi-
```

Now would be a good time to start U-Boot in QEMU and verify that everything works. Start QEMU by passing the built U-Boot binary to it in the `-kernel` parameter, like this (where `u-boot-2018.09` is a subfolder name that you might need to change):

```
qemu-system-arm -M vexpress-a9 -m 32M -no-reboot -nographic -monitor
    telnet:127.0.0.1:1234,server,nowait -kernel u-boot-2018.09/u-boot
```

QEMU should show U-Boot starting up, and if you hit a key when U-Boot prompts `Hit any key to stop autoboot`, you'll be dropped into the U-Boot command line. With that, we can be satisfied that U-Boot was built correctly and works, so next we can tell it to boot something specific, like our program.

Creating a SD card

On a real hardware board, you would probably have U-Boot and your program stored in the program flash. This doesn't comfortably work with QEMU and the Versatile Express series, so we'll take another approach that is very similar to what you could on hardware. We will create a SD card image, place our program there, and tell U-Boot to boot it. What follows is again not particularly related to ARM programming, but rather a convenient way of preparing an image.

First we'll need an additional package that can be installed with `sudo apt-get install qemu-utils`.

Next we need the SD card image itself, which we can create with `qemu-img`. Then we will create an ext2 partition on the SD card, and finally copy the uImage containing our code to the card (we'll create the uImage in the next section). It is not easily possible to manipulate partitions directly inside an image file, so we will need to mount it using `qemu-nbd`, a tool that makes it possible to mount QEMU images as network block devices. The following script, which I called `create-sd.sh`, can be used to automate the process:

```
#!/bin/bash

SDNAME="$1"
UIMGNAME="$2"

if [ "$#" -ne 2 ]; then
    echo "Usage: "$0" sdimage uimage"
    exit 1
fi

command -v qemu-img >/dev/null || { echo "qemu-img not installed"; exit 1; }
command -v qemu-nbd >/dev/null || { echo "qemu-nbd not installed"; exit 1; }

qemu-img create "$SDNAME" 64M
sudo qemu-nbd -c /dev/nbd0 "$SDNAME"
(echo o;
echo n; echo p
echo 1
echo ; echo
echo w; echo p) | sudo fdisk /dev/nbd0
sudo mkfs.ext2 /dev/nbd0p1

mkdir tmp || true
sudo mount -o user /dev/nbd0p1 tmp/
sudo cp "$UIMGNAME" tmp/
sudo umount /dev/nbd0p1
rmdir tmp || true
sudo qemu-nbd -d /dev/nbd0
```

The script creates a 64 megabyte SD card image, mounts it as a network block device, creates a single ext2 partition spanning the entire drive, and copies the supplied uImage to it. From the command line, the script could then be used like

```
./create-sd.sh sdcard.img bare-arm.uimg
```

to create an image called `sdcard.img` and copy the `bare-arm.uimg` uImage onto the emulated SD card (we'll create the image below, running the command at this point will fail).

NOTE

Depending on your system, you might get an error about `/dev/nbd0` being unavailable when you run the SD card creation script. The most likely cause of such an error is that you don't have the `nbd` kernel

module loaded. Loading it with `sudo modprobe nbd` should create `/dev/nbd0`. To permanently add the module to the load list, you can do `echo "nbd" | sudo tee -a /etc/modules`

Creating the uImage

Now that we have created a SD card and can copy an uImage to it, we have to create the uImage itself.

First of all, what is an uImage? The U-Boot bootloader can load applications from different types of images. These images can consist of multiple parts, and be fairly complex, like how Linux gets booted. We are not trying to boot a Linux kernel or anything else complicated, so we'll be using an older image format for U-Boot, which is then the uImage format. The uImage format consists of simply the raw data and a header that describes the image. Such images can be created with the `mkimage` utility, which is part of U-Boot itself. When we built U-Boot, `mkimage` should have been built as well.

Let's call `mkimage` and ask it to create an U-Boot uImage out of the application we had previously, the "better hang" one. From now on, we'll also be able to use ELF files instead of the raw binary dumps because U-Boot knows how to load ELF files. `mkimage` should be located in the `tools` subfolder of the U-Boot folder. Assuming our `better-hang.elf` is still present, we can do the following:

```
1 u-boot-2018.09/tools/mkimage -A arm -C none -T kernel -a 0x60000000 -e
     0x60000000 -d better-hang.elf bare-arm.uimg
```

With that, we say that we want an uncompressed (`-C none`) image for ARM (`-A arm`), the image will contain an OS kernel (`-T kernel`). With `-d better-hang.bin` we tell `mkimage` to put that `.bin` file into the image. We told U-Boot that our image will be a kernel, which is not really true because we don't have an operating system. But the `kernel` image type indicates to U-Boot that the application is not going to return control to U-Boot, and that it will manage interrupts and other low-level things by itself. This is what we want since we're looking at how to do low-level programming in bare metal.

We also indicate that the image should be loaded at `0x60000000` (with `-a`) and that the entry point for the code will be at the same address (with `-e`). This choice of address is because we want to load the image into the RAM of our device, and in the previous chapter we found that RAM starts at `0x60000000` on the board. Is it safe to place our code into the beginning of RAM? Will it not overwrite U-Boot itself and prevent a proper boot? Fortunately, we don't have that complication. U-Boot is initially executed from ROM, and then, on ARM system, it copies itself to the **end** of the RAM before continuing from there.

When the uImage is created, we need to copy it to the SD card. As noted previously, it can be done just by executing `./create-sd.sh sdcard.img bare-arm.uimg` thanks to the script created before. If everything went well, we now have a SD card image that can be supplied to QEMU.

Booting everything

We're ready to boot! Let's start QEMU as usual, except that this time we'll also add an extra parameter telling QEMU that we want to use a SD card.

```
1 qemu-system-arm -M vexpress-a9 -m 32M -no-reboot -nographic -monitor
    telnet:127.0.0.1:1234,server,nowait -kernel u-boot-2018.09/u-boot -
    sd sdcard.img
```

Hit a key when U-Boot prompts you to, in order to use the U-Boot command line interface. We can now use a few commands to examine the state of things and confirm that everything is as we wanted. First type `mmc list` and you should get a response like `MMC: 0`. This confirms the presence of an emulated SD card. Then type `ext2ls mmc 0`. That is the equivalent of running `ls` on the SD card's filesystem, and you should see a response that includes the `bare-arm.uimg` file - our uImage.

Let's load the uImage into memory. We can tell U-Boot to do that with `ext2load mmc 0 0x60000000 bare-arm.uimg`, which as can probably guess means to load `bare-arm.uimg` from the `ext2` system on the first MMC device into address `0x60000000`. U-Boot should report success, and then we can use `iminfo 0x60000000` to verify whether the image is located at that address now. If everything went well, U-Boot should report that a legacy ARM image has been found at the address, along with a bit more information about the image. Now we can go and boot the image from memory: `bootm 0x60000000`.

U-Boot will print `Starting kernel...` and seemingly hang. You can now check the QEMU monitor (recall that you can connect to it with `telnet localhost 1234`), and issue the `info registers` command to see that R2 is once again equal to `0xDEADBEEF`. Success! Our program has now been loaded from a SD card image and started through U-Boot!

Better yet, the modifications we made earlier to U-Boot allow it to perform this boot sequence automatically. With those modifications, we added a new environment variable to U-Boot, `bootcmd_bare_arm`, which contains the boot commands. If you type `printenv bootcmd_bare_arm` in the U-Boot command-line, you'll see the boot sequence.

If you start QEMU again and don't press any keys to pause U-Boot, you should see If you type `printenv bootcmd_bare_arm` in the U-Boot command-line, you'll see the boot sequence. If you start QEMU again and don't press any keys to pause U-Boot, you should see the boot continue automatically.

NOTE

Modifying the U-Boot source code in order to save a command sequence may seem strange, and indeed we're doing that because of QEMU emulation. Normally, running U-Boot from a writable device such as a SD card would let us use U-Boot's `setenv` and `saveenv` commands to permanently save changes without recompiling the whole bootloader.

Having now completed a boot sequence fairly similar to real hardware, we can continue with our own programming. In the next chapter, we'll continue by getting some C code running.

4 Preparing a C environment

In this part, we will do some significant work to get a proper application up and running. What we have done so far is hardly an application, we only execute one instruction before hanging, and everything is done in the reset handler. Also, we haven't written or run any C code so far. Programming in assembly is harder than in C, so generally bare-metal programs will do a small amount of initialization in assembly and then hand control over to C code as soon as possible.

We are now going to write startup code that prepares a C environment and runs the `main` function in C. This will require setting up the stack and also handling data relocation, that is, copying some data from ROM to RAM. The first C code that we run will print some strings to the terminal by accessing the UART peripheral device. This isn't really using a proper driver, but performing some UART prints is a very good way of seeing that things are working properly.

New startup code

Setting up the stack

Our startup code is getting a major rework. It will do several new and exciting things, the most basic of which is to prepare the stack. C code will not execute properly without a stack, which is necessary among other things to have working function calls.

Conceptually, preparing the stack is quite simple. We just pick two addresses in our memory space that will be the start and end of the stack, and then set the initial stack pointer to the start of the stack. By convention, the stack grows towards lower addresses, that is, your stack could be between addresses like `0x60020000` and `0x60021000`, which would give a stack of `0x1000` bytes, and the stack pointer would initially point to the "end" address `0x600210000`. This is called a descending stack. The ARM Procedure Call Standard also specifies that a descending stack should be used.

The ARMv7A architecture has, on a system level, several stack pointers and the CPU has several processor modes. Consulting the ARMv7A reference manual, we can see that processor modes are: user, FIQ, IRQ, supervisor, monitor, abort, undefined and system. To simplify things, we will only care about three modes now - the FIQ and IRQ modes, which execute fast interrupt and normal interrupt code respectively - and the supervisor mode, which is the default mode the processor starts in.

Further consulting the manual (section *B1.3.2 ARM core registers*), we see that the ARM core registers, R0 to R15, differ somewhat depending on the mode. We are now interested in the stack pointer register, SP or R13, and the manual indicates that the "current" SP register an application sees depends on the processor mode, with the supervisor, IRQ and FIQ modes each having their own SP register.

NOTE

Referring to ARM registers, SP and R13 always means the same thing. Similarly, LR is R14 and PC is R15. The name used depends on the documentation or the tool you're using, but don't get confused by the three special registers R13-R15 having multiple names.

In our startup code, we are going to switch to all the relevant modes in order and set the initial stack pointer in the SP register to a memory address that we allocate for the purpose. We'll also fill the stack with a garbage pattern to indicate it's unused.

First we add some defines at the beginning of our `startup.s`, with values for the different modes taken from the ARMv7A manual.

```
1  /* Some defines */
2  .equ MODE_FIQ, 0x11
3  .equ MODE_IRQ, 0x12
4  .equ MODE_SVC, 0x13
```

Now we change our entry code in `Reset_Handler` so that it starts like this:

```
1  Reset_Handler:
2      /* FIQ stack */
3      msr cpsr_c, MODE_FIQ
4      ldr r1, =_fiq_stack_start
5      ldr sp, =_fiq_stack_end
6      movw r0, #0xFEFE
7      movt r0, #0xFEFE
8
9  fiq_loop:
10     cmp r1, sp
11     strlt r0, [r1], #4
12     blt fiq_loop
```

Let's walk through the code in more detail.

```
1       msr cpsr_c, MODE_FIQ
```

One of the most important registers in an ARMv7 CPU is the `CPSR` register, which stands for *Current Program Status Register*. This register can be written with the special `msr` instruction - a regular `mov` will not work. `cpsr_c` is used in the instruction in order to change `CPSR` without affecting the condition flags in bits 28-31 of the `CPSR` value. The least significant five bits of `CPSR` form the mode field, so writing to it is the way to switch processor modes. By writing `0x11` to the `CPSR` mode field, we switch the processor to FIQ mode.

```
1       ldr r1, =_fiq_stack_start
2       ldr sp, =_fiq_stack_end
```

We load the end address of the FIQ stack into `SP`, and the start address into `R1`. Just setting the `SP` register would be sufficient, but we'll use the start address in `R1` in order to fill the stack. The actual addresses for `_fiq_stack_end` and other symbols we're using in stack initialization will be output by the linker with the help of our linkscript, covered later.

```
1       movw r0, #0xFEFE
2       movt r0, #0xFEFE
```

These two lines just write the value `0xFEFEFEFE` to `R0`. ARM assembly has limitations on what values can be directly loaded into a register with the `mov` instruction, so one common way to load a 4-byte constant into a register is to use `movw` and `movt` together. In general `movt r0, x; movw r0, y` corresponds to loading `x << 16 | y` into `R0`.

```
1   fiq_loop:
2       cmp r1, sp
3       strlt r0, [r1], #4
4       blt fiq_loop
```

NOTE

The `strlt` and `blt` assembly instructions you see above are examples of ARM's *conditional execution* instructions. Many instructions have conditional forms, in which the instruction has a condition code

suffix. These instructions are only executed if certain flags are set. The store instruction is normally `str`, and then `strlt` is the conditional form that will only execute if the `lt` condition code is met, standing for less-than. So a simplified way to state things is that `strlt` will only execute if the previous compare instruction resulted in a less-than status.

This is the loop that actually fills the stack with `0xFEFEFEFE`. First it compares the value in R1 to the value in SP. If R1 is less than SP, the value in R0 will be written to the address stored in R1, and R1 gets increased by 4. Then the loop continues as long as R1 is less than SP.

Once the loop is over and the FIQ stack is ready, we repeat the process with the IRQ stack, and finally the supervisor mode stack (see the code in the full listing of `startup.s` at the end).

Handling sections and data

A rundown on sections

To get the next steps right, we have to understand the main segments that a program normally contains, and how they normally appear in ELF file sections.

- The code segment, normally called `.text`. It contains the program's executable code, which normally means it's read-only and has a known size.
- The data segment, normally called `.data`. It contains data that can be modified by the program at runtime. In C terms, global and static variables that have a non-zero initial value will normally go here.
- The read-only data segment, usually with a corresponding section called `.rodata`, sometimes merged with `.text`. Contains data that cannot be modified at runtime, in C terms constants will be stored here.
- The unitialized data segment, also known as the BSS segment and with the corresponding section being `.bss`. Don't worry about the strange name, it has a history dating back to the 1950s. C variables that are static and have no initial value will end up here. The BSS segment is expected to be filled with zeroes, so variables here will end up initialized to zero. Modern compilers are also smart enough to assign variables explicitly initialized with zero to BSS, that is, `static int x = 0;` would end up in `.bss`.

Thinking now about our program being stored in some kind of read-only memory, like the onboard flash memory of a device, we can reason about which sections have to be copied into the RAM of the

device. There's no need to copy `.text` - code can be executed directly from ROM. Same for `.rodata`, constants can be read from the ROM. However, `.data` is for modifiable variables and therefore needs to be in RAM. `.bss` needs to be created in RAM and zeroed out.

Most embedded programs need to take care of ROM-to-RAM data copying. The specifics depend on the hardware and the use case. On larger, more capable single-board computers, like the popular Raspberry Pi series, it's reasonable to copy even the code (`.text` section) to RAM and run from RAM because of the performance benefits as opposed to reading the ROM. On microcontrollers, copying the code to RAM isn't even an option much of the time, as a typical ARM-based microcontroller using a Cortex-M3 CPU or similar might have around 1 megabyte flash memory but only 96 kilobytes of RAM.

New section layout

Previously we had written `linkscript.ld`, a small linker script. Now we will need a more complete linker script that will define the data sections as well. It will also need to export the start and end addresses of sections so that copying code can use them, and finally, the linker script will also export some symbols for the stack, like the `_fiq_stack_start` that we used in our stack setup code.

Normally, we would expect the program to be stored in flash or other form of ROM, as mentioned before. With QEMU it is possible to emulate flash memory on a number of platforms, but unfortunately not on the Versatile Express series. We'll do something different then and pretend that a section of the emulated RAM is actually ROM. Let's say that the area at `0x60000000`, where RAM actually starts, is going to be treated as ROM. And let's then say that we'll use `0x70000000` as the pretend starting point of RAM. There's no need to skip so much memory - the two points are 256 MB apart - but it's then very easy to look at addresses and immediately know if it's RAM from the first digit.

The first thing to do in the linker script, then, is to define the two different memory areas, our (pretend) ROM and RAM. We do this after the `ENTRY` directive, using a `MEMORY` block.

```
1 MEMORY
2 {
3     ROM (rx) : ORIGIN = 0x60000000, LENGTH = 1M
4     RAM (rwx): ORIGIN = 0x70000000, LENGTH = 32M
5 }
```

NOTE

The GNU linker, `ld`, can occasionally appear to be picky with the syntax. In the snippet above, the spaces are also significant. If you write `LENGTH=1M` without spaces, it won't work, and you'll be rewarded with a terse "syntax error" message from the linker.

Our section definitions will now be like this:

```
    .text : {
        startup.o (.vector_table)
        *(.text)
        *(.rodata)
     } > ROM
    _text_end = .;
    .data : AT(ADDR(.text) + SIZEOF(.text))
    {
        _data_start = .;
        *(.data)
        . = ALIGN(8);
        _data_end = .;
    } > RAM
    .bss : {
        _bss_start = .;
        *(.bss)
        . = ALIGN(8);
        _bss_end = .;
    } > RAM
```

The `.text` section is similar to what we had before, but we're also going to append `.rodata` to it to make life easier with one section less. We write `> ROM` to indicate that `.text` should be linked to ROM. The `_text_end` symbol exports the address where `.text` ends in ROM and hence where the next section starts.

`.data` follows next, and we use `AT` to specify the load address as being right after `.text`. We collect all input `.data` sections in our output `.data` section and link it all to RAM, defining `_data_start` and `_data_end` as the RAM addresses where the `.data` section will reside at runtime. These two symbols are written inside the block that ends with `> RAM`, hence they will be RAM addresses.

`.bss` is handled in a similar manner, linking it to RAM and exporting the start and end addresses.

Copying ROM to RAM

As discussed, we need to copy the `.data` section from ROM to RAM in our startup code. Thanks to the linker script, we know where `.data` starts in ROM, and where it should start and end in RAM, which is

all the information we need to perform the copying. In `startup.s`, after dealing with the stacks we now have the following code:

```
    /* Start copying data */
    ldr r0, =_text_end
    ldr r1, =_data_start
    ldr r2, =_data_end

data_loop:
    cmp r1, r2
    ldrlt r3, [r0], #4
    strlt r3, [r1], #4
    blt data_loop
```

We begin by preparing some data in registers. We load `_text_end` into `R0`, with `_text_end` being the address in ROM where `.text` has ended and `.data` starts. `_data_start` is the address in RAM at which `.data` should start, and `_data_end` is correspondingly the end address in RAM. Then the loop itself compares `R1` and `R2` registers and, as long as `R1` is smaller (meaning we haven't reached `_data_end`), we first load 4 bytes of data from ROM into `R3`, and then store these bytes at the memory address in `R1`. The `#4` operand in the load and store instructions ensures we're increasing the values in `R0` and `R1` correspondingly so that loop continues over the entirety of `.data` in ROM.

With that done, we also initialize the `.bss` section with this small snippet:

```
    mov r0, #0
    ldr r1, =_bss_start
    ldr r2, =_bss_end

bss_loop:
    cmp r1, r2
    strlt r0, [r1], #4
    blt bss_loop
```

First we store the value `0` in `R0` and then loop over memory between the addresses `_bss_start` and `_bss_end`, writing `0` to each memory address. Note how this loop is simpler than the one for `.data` - there is no ROM address stored in any registers. This is because there's no need to copy anything from ROM for `.bss`, it's all going to be zeroes anyway. Indeed, the zeroes aren't even stored in the binary because they would just take up space.

Handing over to C

To summarize, to hand control over to C code, we need to make sure the stack is initialized, and that the memory for modifiable variables is initialized as needed. Now that our startup code handles that, there is no additional magic in how C code is started. It's just a matter of calling the `main` function in C. So we just do that in assembly and that's it:

```
    bl main
```

By using the branch-with-link instruction `bl`, we can continue running in case the `main` function returns. However, we don't actually want it to return, as it wouldn't make any sense. We want to continue running our application as long as the hardware is powered on (or QEMU is running), so a bare-metal application will have an infinite loop of some sorts. In case `main` returns, we'll just indicate an error, same as with internal CPU exceptions.

```
    b Abort_Exception

Abort_Exception:
    swi 0xFF
```

And the above branch should never execute.

Into the C

We're finally ready to leave the complexities of linker scripts and assembly, and write some code in good old C. Create a new file, such as `cstart.c` with the following code:

```
#include <stdint.h>

volatile uint8_t* uart0 = (uint8_t*)0x10009000;

void write(const char* str)
{
    while (*str) {
        *uart0 = *str++;
    }
}

int main() {
```

```
13      const char* s = "Hello world from bare-metal!\n";
14      write(s);
15      *uart0 = 'A';
16      *uart0 = 'B';
17      *uart0 = 'C';
18      *uart0 = '\n';
19      while (*s != '\0') {
20          *uart0 = *s;
21          s++;
22      }
23      while (1) {};
24
25      return 0;
26  }
```

The code is simple and just outputs some text through the device's *Universal Asynchronous Receiver-Transmitter* (UART). The next part will discuss writing a UART driver in more detail, so let's not worry about any UART specifics for now, just note that the hardware's UART0 (there are several UARTs) control register is located at `0x10009000`, and that a single character can be printed by writing to that address.

It's pretty clear that the expected output is:

```
1  Hello world from bare-metal!
2  ABC
3  Hello world from bare-metal!
```

with the first line coming from the call to `write` and the other two being printed from `main`.

The more interesting thing about this code is that it tests the stack, the read-only data section and the regular data section. Let's consider how the different sections would be used when linking the above code.

```
1  volatile uint8_t* uart0 = (uint8_t*)0x10009000;
```

The `uart0` variable is global, not constant, and is initialized with a non-zero value. It will therefore be stored in the `.data` section, which means it will be copied to RAM by our startup code.

```
1  const char* s = "Hello world from bare-metal!\n";
```

Here we have a string, which will be stored in `.rodata` because that's how GCC handles most strings.

And the lines printing individual letters, like `*uart0 = 'A';` shouldn't cause anything to be stored in

any of the data sections, they will just be compiled to instructions storing the corresponding character code in a register directly. The line printing `A` will do `mov r2, #0x41` when compiled, with `0x41` or `65` in decimal being the ASCII code for `A`.

Finally, the C code also uses the stack because of the `write` function. If the stack has not been set up correctly, `write` will fail to receive any arguments when called like `write(s)`, so the first line of the expected output wouldn't appear. The stack is also used to allocate the `s` pointer itself, meaning the third line wouldn't appear either.

Building and running

There are a few changes to how we need to build the application. Assembling the startup code in `startup.s` is not going to change:

```
1 arm-none-eabi-as -o startup.o startup.s
```

The new C source file needs to be compiled, and a few special link options need to be passed to GCC:

```
1 arm-none-eabi-gcc -c -nostdlib -nostartfiles -lgcc -o cstart.o cstart.c
```

With `-nostdlib` we indicate that we're not using the standard C library, or any other standard libraries that GCC would like to link against. The C standard library provides the very useful standard functions like `printf`, but it also assumes that the system implements certain requirements for those functions. We have nothing of the sort, so we don't link with the C library at all. However, since `-nostdlib` disables all default libraries, we explicitly re-add `libgcc` with the `-lgcc` flag. `libgcc` doesn't provide standard C functions, but instead provides code to deal with CPU or architecture-specific issues. One such issue on ARM is that there is no ARM instruction for division, so the compiler normally has to provide a division routine, which is something GCC does in `libgcc`. We don't really need it now but include it anyway, which is good practice when compiling bare-metal ARM software with GCC.

The `-nostartfiles` option tells GCC to omit standard startup code, since we are providing our own in `startup.s`.

We're also providing the `-c` switch to stop after the compilation phase. We don't want GCC to use its default linker script, and we'll perform the linking in the next step with `ld`. Later, when defining proper build targets, this will become streamlined.

Linking everything to obtain an ELF file has not undergone any changes, except for the addition of `cstart.o`:

```
arm-none-eabi-ld -T linkscript.ld -o cenv.elf startup.o cstart.o
```

Even though previously we booted a hang example through U-Boot by using an ELF file directly, now we'll need to go back to using a plain binary, which means calling `objcopy` to convert the ELF file into a binary. Why is this necessary? There's an educational aspect and the practical aspect. From the educational side, having a raw binary is much closer to the situation we would have on real hardware, where the on-board flash memory would be written with the raw binary contents. The practical aspect is that U-Boot's support of ELF files is limited. It can load an ELF into memory and boot it, but U-Boot doesn't handle ELF sections correctly, as it doesn't perform any relocation. When loading an ELF file, U-Boot just copies it to RAM and ignores how the sections should be placed. This creates problems starting with the `.text` section, which will not be in the expected memory location because U-Boot retains the ELF file's header and any padding that might exist between it and `.text`. Workarounds for these problems are possible, but using a binary file is simpler and much more reasonable.

We convert `cenv.elf` into a raw binary as follows:

```
arm-none-eabi-objcopy -O binary cenv.elf cenv.bin
```

Finally, when invoking `mkimage` to create the U-Boot image, we specify the binary as the input. After that we can create the SD card image using the same `create-sd.sh` script we made in the previous part.

```
mkimage -A arm -C none -T kernel -a 0x60000000 -e 0x60000000 -d cenv.
    bin bare-arm.uimg
./create-sd.sh sdcard.img bare-arm.uimg
```

That's it for building the application! All that remains is to run QEMU (remember to specify the right path to the U-Boot binary). One change in the QEMU command-line is that we will now use `-m 512M` to provide the machine with 512 megabytes of RAM. Since we're using RAM to also emulate ROM, we need the memory at `0x60000000` to be accessible, but also the memory at `0x70000000`. With those addresses being 256 megabytes apart, we need to tell QEMU to emulate at least that much memory.

```
qemu-system-arm -M vexpress-a9 -m 512M -no-reboot -nographic -monitor
    telnet:127.0.0.1:1234,server,nowait -kernel ../common_uboot/u-boot -
    sd sdcard.img
```

Run QEMU as above, and you should see the three lines written to UART by our C code. There's now a real program running on our ARM Versatile Express!

Bonus: exploring the ELF file

Dealing with linker scripts, ELF sections and relocations can be difficult. One indispensable tool is `objdump`, capable of displaying all kinds of information about object files, as well as disassembling them. Let's look at some of the useful commands to run on our `cenv.elf`. First is the `-h` option, which summarizes sections headers.

```
arm-none-eabi-objdump -h cenv.elf

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000001ea  60000000  60000000  00010000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000008  70000000  600001ea  00020000  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  70000008  600001f2  00020008  2**0
                  ALLOC
```

Of particular interest are the load address (LMA), which indicates where the section would be in ROM, and the virtual address (VMA), which indicates where the section would be during runtime, which in our case means RAM. We can also look at the contents of an individual section. Suppose we want to know what's in `.data`:

```
arm-none-eabi-objdump -s -j .data cenv.elf

Contents of section .data:
 70000000 00900010 00000000                    ........
```

The `70000000` in the beginning is the address, and then we see the actual data - `00900010` can be recognized as the little-endian 4-byte encoding of `0x10009000`, the address of UART0.

Running `objdump` with the `-t` switch shows the symbol table, so for `arm-none-eabi-objdump -t cenv.elf` we would see quite a bit of output, some of it containing:

```
00000011 l       *ABS*  00000000 MODE_FIQ
00000012 l       *ABS*  00000000 MODE_IRQ
00000013 l       *ABS*  00000000 MODE_SVC
60000034 l       .text  00000000 fiq_loop
6000004c l       .text  00000000 irq_loop
600001ea g       .text  00000000 _text_end
70000008 g       .bss   00000000 _bss_start
00001000 g       *ABS*  00000000 _fiq_stack_size
```

```
 9  70000008 g       .bss   00000000 _bss_end
10  600000dc g     F .text  00000054 write
```

You can see how smybols defined in the assembly, C function names and linker-exported symbols are all available in the output.

Finally, running `arm-none-eabi-objdump -d cenv.elf` will disassemble the code, something that usually ends up being necessary at some point in low-level embedded development.

5 Build & debug system

This part is going to be a detour from bare-metal programming in order to quickly set up a build system using CMake, and briefly show how our program can be debugged while running in QEMU. If you're not interested, you can skip this part, though at least skimming it would be recommended.

We will use CMake as our build manager. CMake provides a powerful language to define projects, and doesn't actually build them itself - CMake generates input for another build system, which will be GNU Make since we're developing on Linux. It's also possible to use Make by itself, but for new projects I prefer to use CMake even when its cross-platform capabilities and other powerful features are not needed.

It should also be noted here that I am far from a CMake expert, and build systems aren't the focus of these articles, so this could certainly be done better.

To begin with, we'll organize our project folder with some subfolders, and create a top-level `CMakeLists.txt`, which is the input file CMake normally expects. The better-organized project structure looks like this:

```
|-- CMakeLists.txt
|-- scripts
|    -- create-sd.sh
 -- src
    |-- cstart.c
    |-- linkscript.ld
     -- startup.s
```

The `src` folder contains all the source code, the `scripts` folder is for utility scripts like the one creating our SD card image, and at the top there's `CMakeLists.txt`.

We want CMake to handle the following for us:

- Rebuild U-Boot if necessary
- Build our program, including the binary converstion with `objcopy`
- Create the SD card image for QEMU
- Provide a way of running QEMU

To accomplish that, `CMakeLists.txt` can contain the following:

```
cmake_minimum_required (VERSION 2.8)

set(CMAKE_SYSTEM_NAME Generic)
set(CMAKE_SYSTEM_PROCESSOR arm)
set(CMAKE_CROSSCOMPILING TRUE)

set(UBOOT_PATH "${CMAKE_CURRENT_SOURCE_DIR}/../common_uboot")
set(MKIMAGE "${UBOOT_PATH}/tools/mkimage")

project (bare-metal-arm C ASM)

set(CMAKE_C_COMPILER "arm-none-eabi-gcc")
set(CMAKE_ASM_COMPILER "arm-none-eabi-as")
set(CMAKE_OBJCOPY "arm-none-eabi-objcopy")

file(GLOB LINKSCRIPT "src/linkscript.ld")
set(ASMFILES src/startup.s)
set(SRCLIST src/cstart.c)

set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -nostartfiles -nostdlib -g -Wall")
set(CMAKE_EXE_LINKER_FLAGS "-T ${LINKSCRIPT}")

add_custom_target(u-boot
            COMMAND make vexpress_ca9x4_config ARCH=arm CROSS_COMPILE=
               arm-none-eabi-
            COMMAND make all ARCH=arm CROSS_COMPILE=arm-none-eabi-
            WORKING_DIRECTORY ${UBOOT_PATH})

add_executable(bare-metal ${SRCLIST} ${ASMFILES})
set_target_properties(bare-metal PROPERTIES OUTPUT_NAME "bare-metal.elf
   ")
add_dependencies(bare-metal u-boot)

add_custom_command(TARGET bare-metal POST_BUILD COMMAND ${CMAKE_OBJCOPY
   }
    -O binary bare-metal.elf bare-metal.bin COMMENT "Converting ELF to
       binary")

add_custom_command(TARGET bare-metal POST_BUILD COMMAND ${MKIMAGE}
    -A arm -C none -T kernel -a 0x60000000 -e 0x60000000 -d bare-metal.
       bin bare-arm.uimg
    COMMENT "Building U-Boot image")

add_custom_command(TARGET bare-metal POST_BUILD COMMAND bash ${
   CMAKE_CURRENT_SOURCE_DIR}/scripts/create-sd.sh
    sdcard.img bare-arm.uimg
    COMMENT "Creating SD card image")
```

```
42
43 add_custom_target(run)
44 add_custom_command(TARGET run POST_BUILD COMMAND
45                   qemu-system-arm -M vexpress-a9 -m 512M -no-reboot -
                         nographic
46                   -monitor telnet:127.0.0.1:1234,server,nowait -kernel $
                         {UBOOT_PATH}/u-boot -sd sdcard.img -serial mon:
                         stdio
47                   COMMENT "Running QEMU...")
```

We begin with some preparation, telling CMake that we'll be cross-compiling so it doesn't assume it's building a Linux application, and we store the U-Boot path in a variable. Then there's the specification of the project itself:

```
1 project (bare-metal-arm C ASM)
```

`bare-metal-arm` is an arbitrary project name, and we indicate that C and assembly files are used in it. The next few lines explicitly specify the cross-compiler toolchain elements to be used. After that, we list the source files included in the project:

```
1 file(GLOB LINKSCRIPT "src/linkscript.ld")
2 set(ASMFILES src/startup.s)
3 set(SRCLIST src/cstart.c)
```

Some people prefer to specify wildcards so that all `.c` files are included automatically, but CMake guidelines recommend against this approach. This is a matter of debate, but here I chose to go with explicitly listed files, meaning that new files will need to manually be added here.

CMake lets us specify flags to be passed to the compiler and linker, which we do:

```
1 set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -nostartfiles -nostdlib -g -Wall")
2 set(CMAKE_EXE_LINKER_FLAGS "-T ${LINKSCRIPT}")
```

Those are like what we used previously, except for the addition of `-g -Wall`. The `-g` switch enables generation of debug symbols, which we'll need soon, and `-Wall` enables all compiler warnings and is very good to use as a matter of general practice.

We then define a custom target to build U-Boot, which just runs U-Boot's regular make commands. After that we're ready to define our main target, which we say is an executable that should be built out of source files in the `SRCLIST` and `ASMFILES` variables, and should be called `bare-metal.elf`:

```
add_executable(bare-metal ${SRCLIST} ${ASMFILES})
set_target_properties(bare-metal PROPERTIES OUTPUT_NAME "bare-metal.elf
    ")
```

The two subsequent uses of `add_custom_command` are to invoke `objcopy` and call the `create-sd.sh` script to build the U-Boot uimage.

That's everything we need to build our entire program, all the way to the U-Boot image containing it. It is, however, also convenient to have a way of running QEMU from the same environment, so we also define a target called `run` and provide the QEMU command-line to it. The addition of `-serial mon:stdio` in the QEMU command line means that we'll be able to issue certain QEMU monitor commands directly in the terminal, giving us a cleaner shutdown option.

Building and running

When building the project, I strongly recommend doing what's known as an *out of source build*. It simply means that all the files resulting from the build should go into a separate folder, and not your source folder. This is cleaner (no mixing of source and object files), easier to use with Git (just ignore the whole build folder), and allows you to have several builds at the same time.

To build with CMake, first you need to tell CMake to generate the build configuration from `CMakeLists.txt`. The easiest way to perform a build is:

```
cmake -S . -Bbuild
```

The `-S .` option says to look for the source, starting with `CMakeLists.txt`, in the current folder, and `-Bbuild` specifies that a folder called `build` should contain the generated configuration. After running that command, you'll have the `build` folder containing configurations generated by CMake. You can then use `make` inside that folder. There's no need to call CMake itself again unless the build configuration is supposed to change. If you, for instance, add a new source file to the project, you need to include it in `CMakeLists.txt` and call CMake again.

From the newly created `build` folder, simply invoking the default make target will build everything.

```
make
```

You'll see compilation of U-Boot and our own project, and the other steps culminating in the creation of `sdcard.img`. Since we also defined a CMake target to run QEMU, it can also be invoked directly after building. Just do:

```
make run
```

and QEMU will run our program. Far more convenent than what we had been doing.

HINT

If you run QEMU with `make run` and then terminate it with Ctrl-C, you'll get messages about the target having failed. This is harmless but doesn't look nice. Instead, you can cleanly quit QEMU with Ctrl-A, X (that is Ctrl-A first and then X without the Ctrl key). It's a feature of the QEMU monitor, and works because of adding `-serial mon:stdio` to the QEMU command-line.

Debugging in QEMU with GDB

While the QEMU monitor provides many useful features, it's not a proper debugger. When running software on a PC through QEMU, as opposed to running on real hardware, it would be a waste not to take advantage of the superior debug capabilities available. We can debug our bare-metal program using the GDB, the GNU debugger. GDB provides remote debugging capabilities, with a server called `gdbserver` running on the machine to be debugged, and then the main `gdb` client communicatng with the server. QEMU is able to start an instance of `gdbserver` along with the program it's emulating, so remote debugging is a possibility with QEMU and GDB.

Starting `gdbserver` when running QEMU is as easy as adding `-gdb tcp::2159` to the QEMU command line (2159 is the standard port for GDB remote debugging). Given that we're using CMake, we can use it to define a new target for a debug run of QEMU. These are the additions in `CMakeLists.txt`:

```
string(CONCAT GDBSCRIPT "target remote localhost:2159\n"
                        "file bare-metal.elf")
file(WRITE ${CMAKE_BINARY_DIR}/gdbscript ${GDBSCRIPT})

add_custom_target(drun)
add_custom_command(TARGET drun PRE_BUILD COMMAND ${CMAKE_COMMAND} -E
    cmake_echo_color --cyan
                    "To connect the debugger, run arm-none-eabi-gdb -x
                        gdbscript")
```

```
8  add_custom_command(TARGET drun PRE_BUILD COMMAND ${CMAKE_COMMAND} -E
       cmake_echo_color --cyan
9                     "To start execution, type continue in gdb")
10
11 add_custom_command(TARGET drun POST_BUILD COMMAND
12                    qemu-system-arm -S -M vexpress-a9 -m 512M -no-reboot -
                          nographic -gdb tcp::2159
13                    -monitor telnet:127.0.0.1:1234,server,nowait -kernel ${
                          UBOOT_PATH}/u-boot -sd sdcard.img -serial mon:
                          stdio
14                    COMMENT "Running QEMU with debug server...")
```

The `drun` target (for *debug run*) adds `-gdb tcp::2159` to start `gdbserver`, and `-S`, which tells QEMU not to start execution after loading. That option is useful for debugging because it gives you the time to set breakpoints, letting you debug the code very early if you need to.

When debugging remotely, GDB needs to know what server to connect to, and where to get the debug symbols. We can connect using the GDB command `target remote localost:2159` and then load the ELF file using `file bare-metal.elf`. To avoid typing those commands manually all the time, we ask CMake to put them into a file called `gdbscript` that GDB can read when started.

Let's rebuild and try a quick debug session.

```
1 cmake -S . -Bbuild
2 cd build
3 make
4 make drun
```

You should see CMake print some hints that we provided, and then QEMU will wait doing nothing. In another terminal now, you can start GDB from the `build` folder, telling it to read commands from the `gdbscript` file:

```
1 arm-none-eabi-gdb -x gdbscript
```

If you're using a Linux distribution from 2018 or later (Ubuntu 18.04 or Debian 10 for example), there might be no `arm-none-eabi-gdb`. In that case, run `gdb-multiarch` instead (after installing with `sudo apt-get install gdb-multiarch` if needed).

Now you have GDB running and displaying `(gdb)`, its command prompt. You can set a breakpoint using the **`break`** command, let's try to set one in the `main` function, and then continue execution with `c` (short for **`continue`**):

```
(gdb) break main
(gdb) c
```

As soon as you issue the `c` command, you'll see QEMU running. After U-Boot output, it will stop again, and GDB will show that it hit a breakpoint, something like the following:

```
Breakpoint 1, main () at /some/path/to/repo/src/05_cmake/src/cstart.c
    :13
13              const char* s = "Hello world from bare-metal!\n";
```

From there, you can use the `n` command to step through the source code, `info stack` to see the stack trace, and any other GDB commands.

I won't be covering GDB in additional detail here, that's outside the scope of these tutorials. GDB has a comprehensive, if overwhelming, manual, and there's a lot more material available online. Beej's guide to GDB, authored by Brian Hall, is perhaps the best getting-started guide for GDB. If you'd rather use a graphical front-end, there is also a large selection. When looking for GDB-related information online, don't be alarmed if you find old articles - GDB dates back to the 1980s, and while it keeps getting new features, the basics haven't changed in a very long time.

Our project now has a half-decent build system, we are no longer relying on manual steps, and can debug our program. This is a good place to continue from!

6 UART driver development

This chapter will concern driver development, a crucial part of bare-metal programming. We will walk through writing a UART driver for the Versatile Express series, but the ambition here is not so much to cover that particular UART in detail as it is to show the general approach and patterns when writing a similar driver. As always with programming, there is a lot that can be debated, and there are parts that can be done differently. Starting with a UART driver specifically has its advantages. UARTs are very common peripherals, they're much simpler than other serial buses such as SPI or I2C, and the UART pretty much corresponds to standard input/output when run in QEMU.

Doing the homework

Writing a peripheral driver is not something you should just jump into. You need to understand the device itself, and how it integrates with the rest of the hardware. If you start coding off the hip, you're likely to end up with major design issues, or just a driver that mysteriously fails to work because you missed a small but crucial detail. Thinking before doing should apply to most programming, but driver programming is particularly unforgiving if you fail to follow that rule.

Before writing a peripheral device driver, we need to understand, in broad strokes, the following about the device:

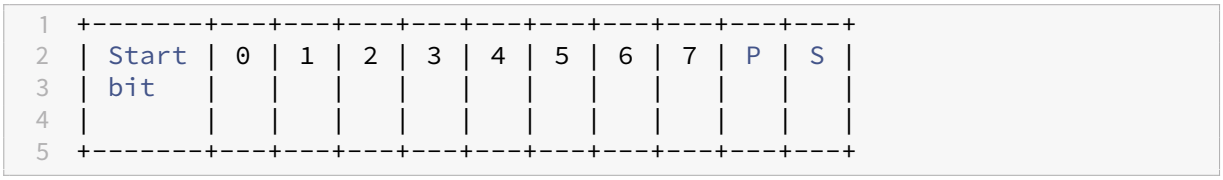
- How it performs its function(s). Whether it's a communication device, a signal converter, or anything else, there are going to be many details of how the device operates. In the case of a UART device, some of the things that fall here are, what baud rates does it support? Are there input and output buffers? When does it sample incoming data?
- How it is controlled. Most of the time, the peripheral device will have several registers, writing and reading them is what controls the device. You need to know what the registers do.
- How it integrates with the hardware. When the device is part of a larger system, which could be a system-on-a-chip or a motherboard-based design, it somehow connects to the rest of the system. Does the device take an external input clock and, if so, where from? Does enabling the device require some other system conditions to be met? The registers for controlling the device are somehow accessible from the CPU, typically by being mapped to a particular memory address.

> From a CPU perspective, registers that control peripherals are *Special Function Registers* (SFR), though not all SFRs correspond to peripherals.

Let's then look at the UART of the Versatile Express and learn enough about it to be ready to create the driver.

Basic UART operation

UART is a fairly simple communications bus. Data is sent out on one wire, and received on another. Two UARTs can thus communicate directly, and there is no clock signal or synchronization of any kind, it's instead expected that both UARTs are configured to use the same baud rate. UART data is transmitted in packets, which always begin with a start bit, followed by 5 to 9 data bits, then optionally a parity bit, then 1 or 2 stop bits. A packet could look like this:



A pair of UARTs needs to use the same frame formatting for successful communication. In practice, the most common format is 8 bits of data, no parity bit, and 1 stop bit. This is sometimes written as `8-N-1` in shorthand, and can be written together with the baud rate like `115200/8-N-1` to describe a UART's configuration.

On to the specific UART device that we have. The Versatile Express hardware series comes with the PrimeCell UART PL011, the reference manual is also available from the ARM website. Reading through the manual, we can see that the PL011 is a typical UART with programmable baud rate and packet format settings, that it supports infrared transmission, and FIFO buffers both for transmission and reception. Additionally there's Direct Memory Access (DMA) support, and support for hardware flow control. In the context of UART, hardware flow control means making use of two additional physical signals so that one UART can inform another when it's ready to send, or ready to receive, data.

Key PL011 registers

The PL011 UART manual also describes the registers that control the peripheral, and we can identify the most important registers that our driver will need to access in order to make the UART work. We will need to work with:

- Data register (DR). The data received, or to be transmitted, will be accessed through DR.

- Receive status / error clear register (RSRECR). Errors are indicated here, and the error flag can also be cleared from here.
- Flag register (FR). Various flags indicating the UART's status are collected in FR.
- Integer baud rate register (IBRD) and Fractional baud rate register (FBRD). Used together to set the baud rate.
- Line control register (LCR_H). Used primarily to program the frame format.
- Control register (CR). Global control of the peripheral, including turning it on and off.

In addition, there are registers related to interrupts, but we will begin by using polling, which is inefficient but simpler. We will also not care about the DMA feature.

As is often the case, reading register descriptions in the manual also reveals some special considerations that apply to the particular hardware. For example, it turns out that writing the IBRD or FBRD registers will not actually have any effect until writing the LCR_H - so even if you only want to update IBRD, you need to perform a sequence of two writes, one to IBRD and another to LCR_H. It is very common in embedded programming to run into such special rules for reading or writing registers, which is one of the reasons reading the manual for the device you're about to program is so important.

PL011 - Versatile Express integration

Now that we are somewhat familiar with the PL011 UART peripheral itself, it's time to look at how integrates with the Versatile Express hardware. The VE hardware itself consists of a motherboard and daughter board, and the UARTs are on the motherboard, which is called the Motherboard Express µATX and of course has its own reference manual.

One important thing from the PL011 manual is the reference clock, UARTCLK. Some peripherals have their own independent clock, but most of them, especially simpler peripherals, use an external reference clock that is then often divided further as needed. For external clocks, the peripheral's manual cannot provide specifics, so the information on what the clock is has to be found elsewhere. In our case, the motherboard's documentation has a separate section on clocks (*2.3 Clock architecture* in the linked PDF), where we can see that UARTs are clocked by the OSC2 clock from the motherboard, which has a frequency of 24 MHz. This is very convenient, we will not need to worry about the reference clock possibly having different values, we can just say it's 24 MHz.

Next we need to find where the UART SFRs are located from the CPU's perspective. The motherboard's manual has memory maps, which differ depending on the daughter board. We're using the CoreTile Express A9x4, so it has what the manual calls the *ARM Legacy memory map* in section 4.2. It says that the address for UART0 is `0x9000`, using SMB (System Memory Bus) chip select CS7, with the chip select introducing an additional offset that the daughter board defines. Then it's on to the CoreTile Express

A9x4 manual, which explains that the board's memory controller places most motherboard peripherals under CS7, and in *3.2 Daughterboard memory map* we see that CS7 memory mappings for accessing the motherboard's peripherals start at `0x10000000`. Thus the address of UART0 from the perspective of the CPU running our code is CS7 base `0x10000000` plus an offset of `0x9000`, so `0x10009000` is ultimately the address we need.

Yes, this means that we have to check two different manuals just to find the peripheral's address. This is, once again, nothing unusual in an embedded context.

Writing the driver

What's in the box?

In higher-level programming, you can usually treat drivers as a black box, if you even give them any consideration. They're there, they do things with hardware, and they only have a few functions you're exposed to. Now that we're writing a driver, we have to consider what it consists of, the things we need to implement. Broadly, we can say that a driver has:

- An initialization function. It starts the device, performing whatever steps are needed. This is usually relatively simple.
- Configuration functions. Most devices can be configured to perform their functions differently. For a UART, programming the baud rate and frame format would fall here. Configuration can be simple or very complex.
- Runtime functions. These are the reason for having the driver in the first place, the interesting stuff happens here. In the case of UART, this means functions to transmit and read data.
- A deinitialization function. It turns the device off, and is quite often omitted.
- Interrupt handlers. Most peripherals have some interrupts, which need to be handled in special functions called interrupt handlers, or interrupt service routines. We won't be covering that for now.

Now we have a rough outline of what we need to implement. We will need code to start and configure the UART, and to send and receive data. Let's get on with the implementation.

Exposing the SFRs

We know by now that programming the UART will be done by accessing the SFRs. It is possible, of course, to access the memory locations directly, but a better way is to define a C struct that reflects

what the SFRs look like. We again refer to the PL011 manual for the register summary. It begins like this:

Table 3-1 Register summary

Offset	Type	Width	Reset value	Name	Description
0x000	RW	12/8	0x---	UARTDR	*Data register, UARTDR* on page 3-5
0x004	RW	4/0	0x0	UARTRSR/ UARTECR	*Receive status register/error clear register, UARTRSR/UARTECR* on page 3-6
0x008-0x014	-	-	-	-	Reserved
0x018	RO	9	0b-10010---	UARTFR	*Flag register, UARTFR* on page 3-8
0x01C	-	-	-	-	Reserved
0x020	RW	8	0x00	UARTILPR	*IrDA low-power counter register, UARTILPR* on page 3-9
0x024	RW	16	0x0000	UARTIBRD	*Integer baud rate register, UARTIBRD* on page 3-10
0x028	RW	6	0x00	UARTFBRD	*Fractional baud rate register, UARTFBRD* on page 3-10
0x02C	RW	8	0x00	UARTLCR_H	*Line control register, UARTLCR_H* on page 3-12
0x030	RW	16	0x0300	UARTCR	*Control register, UARTCR* on page 3-15

Figure 6.1: PL011 register summary

Looking at the table, we can define it in code as follows:

```
typedef volatile struct __attribute__((packed)) {
    uint32_t DR;                /* 0x0 Data Register */
    uint32_t RSRECR;            /* 0x4 Receive status / error clear
       register */
    uint32_t _reserved0[4];     /* 0x8 - 0x14 reserved */
    const uint32_t FR;          /* 0x18 Flag register */
    uint32_t _reserved1;        /* 0x1C reserved */
    uint32_t ILPR;              /* 0x20 Low-power counter register */
    uint32_t IBRD;              /* 0x24 Integer baudrate register */
    uint32_t FBRD;              /* 0x28 Fractional baudrate register */
    uint32_t LCRH;              /* 0x2C Line control register */
    uint32_t CR;                /* 0x30 Control register */
} uart_registers;
```

There are several things to note about the code. One is that it uses fixed-size types like `uint32_t`. Since the C99 standard was adopted, C has included the `stdint.h` header that defines exact-width

integer types. So `uint32_t` is guaranteed to be a 32-bit type, as opposed to `unsigned int`, for which there is no guaranteed fixed size. The layout and size of the SFRs is fixed, as described in the manual, so the struct has to match in terms of field sizes.

For the same reason, `__attribute__((packed))` is provided. Normally, the compiler is allowed to insert padding between struct fields in order to align the whole struct to some size suited for the architecture. Consider the following example:

```
typedef struct {
    char a; /* 1 byte */
    int b;  /* 4 bytes */
    char c; /* 1 byte */
} example;
```

If you compile that struct for a typical x86 system where `int` is 4 bytes, the compiler will probably try to align the struct to a 4-byte boundary, and align the individual members to such a boundary as well, inserting 3 bytes after `a` and another 3 bytes after `c`, giving the struct a total size of 12 bytes.

When working with SFRs, we definitely don't want the compiler to insert any padding bytes or take any other liberties with the code. `__attribute__((packed))` is a GCC attribute (also recognized by some other compilers like clang) that tells the compiler to use the struct as it is written, using the least amount of memory possible to represent it. Forcing structs to be packed is generally not a great idea when working with "normal" data, but it's very good practice for structs that represent SFRs.

Sometimes there might be reserved memory locations between various registers. In our case of the PL011 UART, there are reserved bytes between the `RSRECR` and `FR` registers, and four more after `FR`. There's no general way in C to mark such struct fields as unusable, so giving them names like `_reserved0` indicates the purpose. In our struct definition, we define `uint32_t _reserved0[4];` to skip 16 bytes, and `uint32_t _reserved1;` to skip another 4 bytes later.

Some SFRs are read-only, like the `FR`, in which case it's helpful to declare the corresponding field as `const`. Attempts to write a read-only register would fail anyway (the register would remain unchanged), but marking it as `const` lets the compiler check for attempts to write the register.

Having defined a struct that mimics the SFR layout, we can create a static variable in our driver that will point to the UART0 device:

```
static uart_registers* uart0 = (uart_registers*)0x10009000u;
```

A possible alternative to the above would be to declare a macro that would point to the SFRs, such as `#define UART0 ((uart_registers*)0x10009000u)`. That choice is largely a matter of preference.

Register access width

An important, but easy to overlook, aspect of writing to SFRs is access width. A device might require its SFRs to be written all at once, with one write instruction, or the device might introduce other constraints. The corresponding device manual should indicate how it expects the registers to be accessed. Commonly encountered types of access are:

- Word access. The access operation should have the same width as the machine word, e.g. 32 bits on a 32-bit system. A SFR is usually the size of one word.
- Half-word access. As the name indicates, this means accessing half of a word at a time, so 16 bits on a 32-bit system.
- Byte access. Any byte within the register can be written individually.

Requiring word access, possibly with allowing half-word access, is common. Allowing byte access to SFRs is somewhat less common.

What does this mean in practice, and how to make sure access is of the right width? You have to be aware of how your code is going to access SFRs, consider the following:

```
1 sfr->SOMEFIELD |= 0xFu;
```

Most likely, the line would result in assembly code that performs a whole-word write, such as, if the address of `SOMEFIELD` is in register `R0`

```
1 ldr r1, [r0] ; load value in SOMEFIELD into R1
2 orr r2, r1, #15 ; save SOMEFIELD | 0xF into R2
3 str r2, [r0] ; write back to SOMEFIELD
```

or other optimized code that would be even better. However, it's possible that the line would be compiled into code that performs multiple accesses, i.e. one for each byte of SOMEFIELD.

A single word-wide write can be ensured by explicitly asking for a write to a `uint32_t*` such as:

```
1 *(uint32_t*)sfr->SOMEFIELD = some_val;
```

This is not particularly important for the PL011 UART specifically, which does not specify any restrictions on register access, but using word access explicitly would be good practice nonetheless. In the next chapter, dealing with the interrupt controller, register access width becomes more important.

Initializing and configuring the UART

Let's now write `uart_configure()`, which will initialize and configure the UART. For some drivers you might want a separate `init` function, but a `uart_init()` here wouldn't make much sense, the device is quite simple. The functionitself is not particularly complex either, but can showcase some patterns.

First we need to define a couple of extra types. For the return type, we want something that can indicate failure or success. It's very useful for functions to be able to indicate success or failure, and driver functions can often fail in many ways. Protecting against possible programmer errors is of particular interest - it's definitely possible to use the driver incorrectly! So one of the approaches is to define error codes for each driver (or each driver type perhaps), like the following:

```
typedef enum {
        UART_OK = 0,
        UART_INVALID_ARGUMENT_BAUDRATE,
        UART_INVALID_ARGUMENT_WORDSIZE,
        UART_INVALID_ARGUMENT_STOP_BITS,
        UART_RECEIVE_ERROR,
        UART_NO_DATA
} uart_error;
```

A common convention is to give the success code a value of `0`, and then we add some more error codes to the enumeration. Let's use this `uart_error` type as the return type for our configuration function.

Then we need to pass some configuration to the driver, in order to set the baud rate, word size, etc. One possibility is to define the following struct describing a config:

```
typedef struct {
    uint8_t     data_bits;
    uint8_t     stop_bits;
    bool        parity;
    uint32_t    baudrate;
} uart_config;
```

This approach, of course, dictates that `uart_configure` would take a parameter of the `uart_config` type, giving us:

```
uart_error uart_configure(uart_config* config)
```

There are other possible design choices. You could omit the struct, and just pass in multiple parameters, like `uart_configure(uint8_t data_bits, uint8_t stop_bits, bool parity, unit32_t baudrate)`. I prefer a struct because those values logically belong together. Yet another option would be to have separate functions per parameter, such as `uart_set_baudrate` and `uart_set_data_bits`, but I think that is a weaker choice, as it can create issues with the order in which those functions are called.

On to the function body. You can see the entire source in the corresponding file for this chapter, and here I'll go through it block by block.

First, we perform some validation of the configuration, returning the appropriate error code if some parameter is outside the acceptable range.

```
    if (config->data_bits < 5u || config->data_bits > 8u) {
        return UART_INVALID_ARGUMENT_WORDSIZE;
    }
    if (config->stop_bits == 0u || config->stop_bits > 2u) {
        return UART_INVALID_ARGUMENT_STOP_BITS;
    }
    if (config->baudrate < 110u || config->baudrate > 460800u) {
        return UART_INVALID_ARGUMENT_BAUDRATE;
    }
```

UART only allows 5 to 8 bits as the data size, and the only choices for the stop bit is to have one or two. For the baudrate check, we just constrain the baudrate to be between two standard values.

With validation done, the rest of the function essentially follows the PL011 UART's manual for how to configure it. First the UART needs to be disabled, allowed to finish an ongoing transmission, if any, and its transmit FIFO should be flushed. Here's the code:

```
    /* Disable the UART */
    uart0->CR &= ~CR_UARTEN;
    /* Finish any current transmission, and flush the FIFO */
    while (uart0->FR & FR_BUSY);
    uart0->LCRH &= ~LCRH_FEN;
```

Having a similar `while` loop is common in driver code when waiting on some hardware process. In this case, the PL011's `FR` has a BUSY bit that indicates if a transmission is ongoing. Setting the FEN bit in `LCRH` to `0` is the way to flush the transmit queue.

What about all those defines like `CR_UARTEN` in the lines above though? Here they are from the corresponding header file:

```
1 #define FR_BUSY          (1 << 3u)
2 #define LCRH_FEN         (1 << 4u)
3 #define CR_UARTEN        (1 << 0u)
```

Typically, one SFR has many individual settings, with one setting often using just one or two bits. The locations of the bits are always in the corresponding manual, but using them directly doesn't make for the most readable code. Consider `uart0->CR &= ~(1u)` or `while (uart0->FR & (1 << 3u))`. Any time you read such a line, you'd have to refer to the manual to check what the bit or mask means. Symbolic names make such code much more readable, and here I use the pattern of `SFRNAME_BITNAME`, so `CR_UARTEN` is the bit called `UARTEN` in the `CR` SFR. I won't include more of those defines in this chapter, but they're all in the full header file.

NOTE

Bit manipulation is usually a very important part of driver code, such as the above snippet. Bitwise operators and shift operators are a part of C, and I won't be covering them here. Hopefully you're familiar enough with bit manipulation to read the code presented here. Just in case though, this cheat sheet might be handy:

Assuming that `b` is one bit,

`x |= b` sets bit `b` in `x`

`x &= ~b` clears bit `b` in `x`

`x & b` checks if `b` is set

One bit in position `n` can be conveniently written as `1` left-shifted `n` places. E.g. bit 4 is `1 << 4` and bit 0 is `1 << 0`

Next we configure the UART's baudrate. This is another operation that translates to fairly simple code, but requires a careful reading of the manual. To obtain a certain baudrate, we need to divide the (input) reference clock with a certain divisor value. The divisor value is stored in two SFRs, `IBRD` for the integer part and `FBRD` for the fractional part. Accordig to the manual, `baudrate divisor = reference clock / (16 * baudrate)`. The integer part of that result is used directly, and the fractional part needs to be converted to a 6-bit number `m`, where `m = integer((fractional part * 64)+ 0.5)`. We can translate that into C code as follows:

```
    double intpart, fractpart;
    double baudrate_divisor = (double)refclock / (16u * config->
        baudrate);
    fractpart = modf(baudrate_divisor, &intpart);

    uart0->IBRD = (uint16_t)intpart;
    uart0->FBRD = (uint8_t)((fractpart * 64u) + 0.5);
```

It's possible to obtain the fractional part with some arithmetics, but we can just use the standard C `modf` function that exists for that purpose and is available after including `<math.h>`. While we cannot use the entire C standard library on bare-metal without performing some extra work, mathematical functions do not require anything extra, so we can use them.

Since our reference clock is 24 MHz, as we established before, the `refclock` variable is `24000000u`. Assuming that we want to set a baudrate of `9600`, first the `baudrate_divisor` will be calculated as `24000000 / (16 * 9600)`, giving `156.25`. The `modf` function will helpfully set `intpart` to `156` and `fractpart` to `0.25`. Following the manual's instructions, we directly write the `156` to `IBRD`, and convert `0.25` to a 6-bit number. `0.25 * 64 + 0.5` is `16.5`, we only take the integer part of that, so `16` goes into `FBRD`. Note that `16` makes sense as a representation of `0.25` if you consider that the largest 6-bit number is `63`.

We continue now by setting up the rest of the configuration - data bits, parity and the stop bit.

```
    uint32_t lcrh = 0u;

    /* Set data word size */
    switch (config->data_bits)
    {
    case 5:
        lcrh |= LCRH_WLEN_5BITS;
        break;
    case 6:
        lcrh |= LCRH_WLEN_6BITS;
        break;
    case 7:
        lcrh |= LCRH_WLEN_7BITS;
        break;
    case 8:
        lcrh |= LCRH_WLEN_8BITS;
        break;
    }

    /* Set parity. If enabled, use even parity */
    if (config->parity) {
        lcrh |= LCRH_PEN;
        lcrh |= LCRH_EPS;
```

```
24          lcrh |= LCRH_SPS;
25      } else {
26          lcrh &= ~LCRH_PEN;
27          lcrh &= ~LCRH_EPS;
28          lcrh &= ~LCRH_SPS;
29      }
30
31      /* Set stop bits */
32      if (config->stop_bits == 1u) {
33          lcrh &= ~LCRH_STP2;
34      } else if (config->stop_bits == 2u) {
35          lcrh |= LCRH_STP2;
36      }
37
38      /* Enable FIFOs */
39      lcrh |= LCRH_FEN;
40
41      uart0->LCRH = lcrh;
```

That is a longer piece of code, but there's not much remarkable about it. For the most part it's just picking the correct bits to set or clear depending on the provided configuration. One thing to note is the use of the temporary `lcrh` variable where the value is built, before actually writing it to the `LCRH` register at the end. It is sometimes necessary to make sure an entire register is written at once, in which case this is the technique to use. In the case of this particular device, `LCRH` can be written bit-by-bit, but writing to it also triggers updates of `IBRD` and `FBRD`, so we might as well avoid doing that many times.

At the end of the above snippet, we enable FIFOs for potentially better performance, and write the `LCRH` as discussed. After that, everything is configured, and all that remains is to actually turn the UART on:

```
1       uart0->CR |= CR_UARTEN;
```

Read and write functions

We can start the UART with our preferred configuration now, so it's a good time to implement functions that actually perform useful work - that is, send and receive data.

Code for sending is very straightforward:

```
1   void uart_putchar(char c) {
2       while (uart0->FR & FR_TXFF);
```

```
    uart0->DR = c;
}

void uart_write(const char* data) {
    while (*data) {
        uart_putchar(*data++);
    }
}
```

Given any string, we just output it one character at a time. The `TXFF` bit in `FR` that `uart_putchar()` waits for indicates a full transmit queue - we just wait until that's no longer the case.

These two functions have **`void`** return type, they don't return `uart_error`. Why? It's again a design decision, meaning you could argue against it, but the write functions don't have any meaningful way of detecting errors anyway. Data is sent out on the bus and that's it. The UART doesn't know if anybody's receiving it, and it doesn't have any error flags that are useful when transmitting. So the **`void`** return type here is intended to suggest that the function isn't capable of providing any useful information regarding its own status.

The data reception code is a bit more interesting because it actually has error checks:

```
uart_error uart_getchar(char* c) {
    if (uart0->FR & FR_RXFE) {
        return UART_NO_DATA;
    }

    *c = uart0->DR & DR_DATA_MASK;
    if (uart0->RSRECR & RSRECR_ERR_MASK) {
        /* The character had an error */
        uart0->RSRECR &= RSRECR_ERR_MASK;
        return UART_RECEIVE_ERROR;
    }
    return UART_OK;
}
```

First it checks if the receive FIFO is empty, using the `RXFE` bit in `FR`. Returning `UART_NO_DATA` in that case tells the user of this code not to expect any character. Otherwise, if data is present, the function first reads it from the data register `DR`, and then checks the corresponding error status - it has to be done in this order, once again according to the all-knowing manual. The PL011 UART can distinguish between several kinds of errors (framing, parity, break, overrun) but here we treat them all the same, using `RSRECR_ERR_MASK` as a bitmask to check if any error is present. In that case, a write to the `RSRECR` register is performed to reset the error flags.

Putting it to use

We need some code to make use of our new driver! One possibility is to rewrite `cstart.c` like the following:

```
#include <stdint.h>
#include <stdbool.h>
#include <string.h>
#include "uart_pl011.h"

char buf[64];
uint8_t buf_idx = 0u;

static void parse_cmd(void) {
    if (!strncmp("help\r", buf, strlen("help\r"))) {
        uart_write("Just type and see what happens!\n");
    } else if (!strncmp("uname\r", buf, strlen("uname\r"))) {
        uart_write("bare-metal arm 06_uart\n");
    }
}

int main() {
        uart_config config = {
            .data_bits = 8,
            .stop_bits = 1,
            .parity = false,
            .baudrate = 9600
        };
        uart_configure(&config);

        uart_putchar('A');
        uart_putchar('B');
        uart_putchar('C');
        uart_putchar('\n');

        uart_write("I love drivers!\n");
        uart_write("Type below...\n");

        while (1) {
            char c;
            if (uart_getchar(&c) == UART_OK) {
                uart_putchar(c);
                buf[buf_idx % 64] = c;
                buf_idx++;
                if (c == '\r') {
                    uart_write("\n");
                    buf_idx = 0u;
                    parse_cmd();
                }
```

```
45                 }
46             }
47
48             return 0;
49     }
```

The `main` function asks the UART driver to configure it for `9600/8-N-1`, a commonly used mode, and then outputs some text to the screen much as the previous chapter's example did. Some more interesting things happen within the **`while`** loop now though - it constantly polls the UART for incoming data and appends any characters read to a buffer, and prints that same character back to the screen. Then, when a carriage return (`\r`) is read, it calls `parse_cmd()`.That's a very basic method of waiting for something to be input and reacting on the Enter key.

`parse_cmd()` is a simple function that has responses in case the input line was `help` or `uname`. This way, without writing anything fancy, we grant our bare-metal program the ability to respond to user input!

Doing a test run

To build our program with the new driver, it needs to be added to the source file list in CMake, and we need to change a couple of flags as well. To add the driver file, just add the file to the appropriate list in `CMakeLists.txt`:

```
1  set(SRCLIST src/cstart.c src/uart_pl011.c)
```

We are using the C standard library now, and we need to link against `libm`, so the compiler and linker flags should now look like this:

```
1  set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -nostartfiles -g -Wall")
2  set(CMAKE_EXE_LINKER_FLAGS "-T ${LINKSCRIPT} -lgcc -lm")
```

When building C programs with GCC, the `-lm` flag is necessary if the program uses mathematical functions declared in the standard header `math.h`. For the rest of the standard library, no separate flag is needed. This special treatment of `libm` stems from technical decisions made a long time ago, and is by now best treated simply as a bit of legacy to remember.

Rebuild everything (since we changed `CMakeLists.txt` that means you also need to invoke `cmake`), run the program in QEMU and, after some output, you should be able to type into the terminal and see what you type. That's the UART driver at work! Each character you type gets returned by

`uart_getchar` and then output to the screen by `uart_putchar`. Try writing `help` and hitting the Enter key - you should see the output as defined in `parse_cmd`.

Unfortunately, this is also the first point where using QEMU is a disadvantage. QEMU emulation of devices has limitations, and the PL011 UART is no exception. Our driver will work in QEMU no matter the baudrate. No matter if we connect the UART to standard input/output (as now) or a Linux character device, the baudrate will not matter. There's also no enable/disable mechanism for the emulated UART - the driver would work even if we never enabled the device in the `CR` register.

There's no real way around those issues short of running on real hardware, the best we can do is try and write the driver as if for the actual hardware device.

Summary

We've written our first driver that interfaces with the hardware directly, and we wrote some proof-of-concept code making use of the driver. A big takeaway is that carefully reading the manual is at least half the work involved in writing a driver. In writing the actual driver code, we also saw how it can be quite different from most non-driver code. There's a lot of bit manipulation, and operations that are order-sensitive in ways that may not be intuitive. The fact that the same location in memory, like the `DR` SFR, acts differently when being read versus written is also something rarely encountered outside of driver code.

Unsurprisingly, the driver written in this chapter is not perfect. Some possibilities for improvement:

- Interrupt handling. Currently the driver is being used in polling mode, constantly asking it if new characters have been received. In most practical cases, polling is too inefficient and interrupts are desired. This is something that the next chapter handles.
- More robustness. Error handling, sanity checks and other measures preventing the driver from being incorrectly are good! The driver could return different error codes for different types of receive errors instead of lumping them all together. The driver could keep track of its own status and prevent functions like `uart_write` from executing before the configuration has been done with `uart_configure`.
- The reference clock is hardcoded as 24 MHz now, the driver should instead query the hardware to find the reference clock's frequency.

7 Interrupts

There's no reasonable way of handling systems programming, such as embedded development or operating system development, without interrupts being a major consideration.

What's an interrupt, anyway? It's a kind of notification signal that a CPU receives as an indication that something important needs to be handled. An interrupt is often sent by another hardware device, in which case that's a *hardware interrupt*. The CPU responds to an interrupt by interrupting its current activity (hence the name), and switching to a special function called an *interrupt handler* or an *interrupt service routine* - ISR for short. After dealing with the interrupt, the CPU will resume whatever it was doing previously.

There are also *software interrupts*, which can be triggered by the CPU itself upon detecting an error, or may be possible for the programmer to trigger with code.

Interrupts are used primarily for performance reasons. A polling-based approach, where external devices are continuously asked for some kind of status, are inefficient. The UART driver we wrote in the previous chapter is a prime example of that. We use it to let the user send data to our program by typing, and typing is far slower than the frequency at which the CPU can check for new data. Interrupts solve this problem by instead letting the device notify the CPU of an event, such as the UART receiving a new data byte.

If you read the manual for the PL011 UART in the previous chapter, you probably remember seeing some registers that control interrupt settings, which we ignored at the time. So, changing the driver to work with interrupts should be just a matter of setting some registers to enable interrupts and writing an ISR to handle them, right?

No, not even close. Inerrupt handling is often quite complicated, and there's work to be done before any interrupts can be used at all, and then there are additional considerations for any ISRs. Let's get to it.

Interrupt handling in ARMv7-A

Interrupt handling is very hardware-dependent. We need to look into the general interrupt handling procedure of the particular architecture, and then into specifics of a particular implementation like a

specific CPU. The ARMv7-A manual provides quite a lot of useful information about interrupt handling on that architecture.

ARMv7-A uses the generic term *exception* to refer, in general terms, to interrupts and some other exception types like CPU errors. An interrupt is called an *IRQ exception* in ARMv7-A, so that's the term the manual names a lot.When an ARMv7-A CPU takes an exception, it transfers control to an instruction located at the appropriate location in the vector table, depending on the exception type. The very first code we wrote for startup began with the vector table.

As a reminder:

```
1 _Reset:
2     b Reset_Handler
3     b . /* 0x4  Undefined Instruction */
4     b . /* 0x8  Software Interrupt */
5     b . /* 0xC  Prefetch Abort */
6     b . /* 0x10 Data Abort */
7     b . /* 0x14 Reserved */
8     b . /* 0x18 IRQ */
```

In the code above, we had an instruction at offset `0x0`, for the reset exception, and dead loops for the other exception types, including IRQs at `0x18`. So normally, an ARMv7-A CPU will execute the instruction at `0x18` starting from the vector table's beginning when it takes an IRQ exception.

There's more that happens, too. When an IRQ exception is taken, the CPU will switch its mode to the IRQ mode, affecting how some registers are seen by the CPU. When we were initially preparing the stack for the C environment, we set several stacks up for different CPU modes, IRQ mode being one of them.

At this point it's worth noting that IRQ (and FIQ) exceptions can be disabled or enabled globally. The `CPSR` register, which you might recall we used to explicitly switch to different modes in Chapter 4, also holds the `I` and `F` bits that control whether IRQs and FIQs are enabled respectively.

Ignoring some advanced ARMv7 features like monitor and hypervisor modes, the sequence upon taking an IRQ exception is the following:

1. Figure out the address of the next instruction to be executed after handling the interrupt, and write it into the `LR` register.
2. Save the `CPSR` register, which contains the current processor status, into the `SPSR` register.
3. Switch to IRQ mode, by changing the mode bits in `CPSR` to `0x12`.
4. Make some additional changes in `CPSR`, such as clearing conditional execution flags.

5. Check the VE (Interrupt Vectors Enabled) bit in the SCTLR (system control register). If VE is 0, go to the start of the vector table plus 0x18. If VE is 1, go to the appropriate implementation-defined location for the interrupt vector.

That last part sounds confusing. What's with that implementation-defined location?

Remember that ARMv7-A is not a CPU. It's a CPU architecture. In this architecture, interrupts are supported as just discussed, and there's always the possibility to use an interrupt handler at 0x18 bytes into the vector table. That is, however, not always convenient. Consider that there can be many different interrupt sources, while the vector table can only contain one branch instruction at 0x18. This means that the the function taking care of interrupts would first have to figure out which interrupt was triggered, and then act appropriately. Such an approach puts extra burden on the CPU as it has to check all possible interrupt sources.

The solution to that is known as vectored interrupts. In a vectored interrupt system, each interrupt has its own vector (a unique ID). Then some kind of vectored interrupt controller is in place that knows which ISR to route each interrupt vector to.

The ARMv7-A architecture has numerous implementations, as in specific CPUs. The architecture description says that vectored interrupts may be supported, but the details are left up to the implementation. The choice of which interrupt system to use, though, is controlled by the architecture-defined SCTLR register. In our case, implementation-defined will mean that vectored interrupts are not supported - the CPU we're using doesn't allow vectored interrupts.

Generic Interrupt Controller of the Cortex-A9

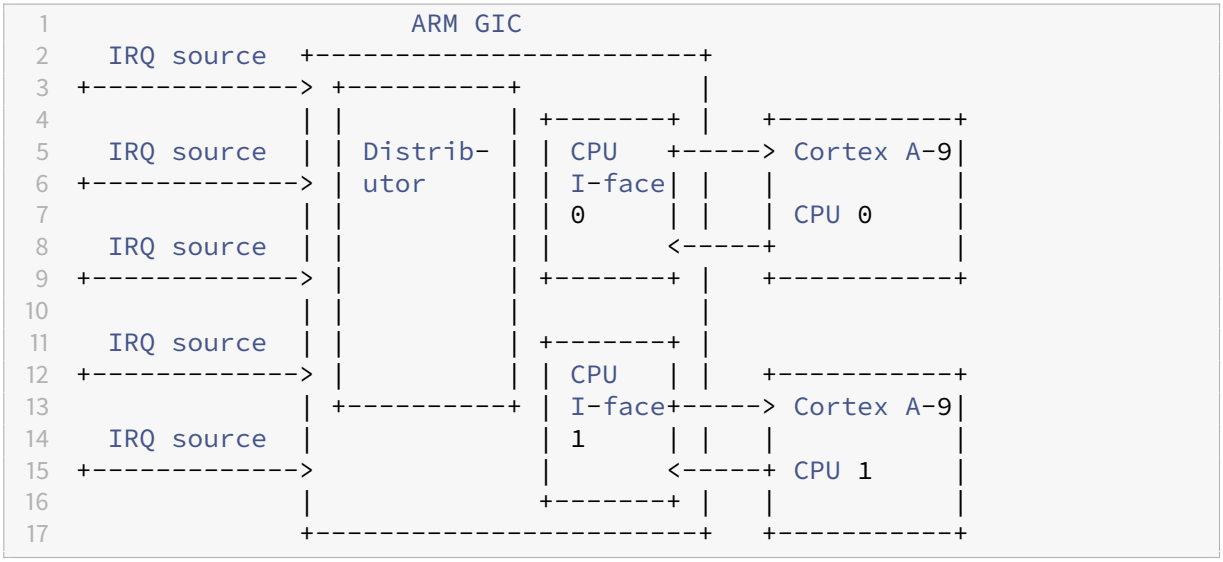
We're programming for a CoreTile Express A9x4 daughterboard, which contains the Cortex-A9 MPCore CPU. The MPCore means it's a CPU that can consist of one to four individual Cortex-A9 cores. So it's the Cortex-A9 MPCore manual that becomes our next stop. There's a chapter in the manual for the interrupt controller - so far so good - but it immediately refers to another manual. Turns out that the Cortex-A9 has an interrupt controller of the *ARM Generic Interrupt Controller* type, for which there's a separate manual (note that GIC version 4.0 makes a lot of references to the ARMv8 architecture). The Cortex-A9 manual refers to version 1.0 of the GIC specification, but reading version 2.0 is also fine, there aren't too many differences and none in the basic features.

The GIC is one of the major interrupt controller implementations. This is one of the area where the difference between A-profile and R-profile of ARMv7 matters. ARMv7-R CPUs such as the Cortex-R4 normally use a vectored controller like the appropriately named VIC.

The GIC has its own set of SFRs that control its operation, and the GIC as a whole is responsible for forwarding interrupt requests to the correct A9 core in the A9-MPCore. There are two main components

in the GIC - the Distributor and the CPU interfaces. The Distributor receives interrupt requests, prioritizes them, and forwards them to the CPU interfaces, each of which corresponds to an A9 core.

Let's clarify with a schematic drawing. The Distributor and the CPU interfaces are all part of the GIC, with each CPU then using its own assigned CPU interface to communicate with the GIC. The communication is two-way because CPUs need to not only receive interrupts but also, at least, to inform the GIC when interrupt handling completes.



To enable interrupts, we'll need to program the GIC Distributor, telling it to enable certain interrupts, and forward them to our CPU. Once we have some form of working interrupt handling, we'll need to tell our program to report back to the GIC, using the CPU Interface part, when the handling of an interrupt has been finished.

The general sequence for an interrupt is as follows:

1. The GIC receives an interrupt request. That particular interrupt is now considered *pending*.
2. If the specific interrupt is enabled in the GIC, the Distributor determines the core or cores to forward it to.
3. Among all pending interrupts, the Distributor chooses the one with the highest priority for each CPU interface.
4. The GIC's CPU interface forwards the interrupt to the processor, if priority rules tell it to do so.
5. The processor acknowledges the interrupt, informing the GIC. The interrupt is now *active* or, possibly, *active and pending* if the interrupt has been requested again.

6. The software running on the processor handles the interrupt and then informs the GIC that the handling is complete. The interrupt is now *inactive*.

Note that interrupts can also be preempted, that is, a higher-priority interrupt can be forwarded to a CPU while it's already processing an active lower-priority interrupt.

Just as with the UART driver previously, it's wise to identify some key registers of the GIC that we will need to program to process interrupts. I'll once again omit the `GIC` prefix in register names for brevity. Registers whose names start with `D` (or `GICD` in full) belong to the Distributor system, those with `C` names belong to the CPU interface system.

For the Distributor, key registers include:

- DCTLR - the global Distributor Control Register, containing the enable bit - no interrupts will be forwarded to CPUs without turning that bit on.
- DISENABLERn - interrupt set-enable registers. There are multiple such registers, hence the n at the end. Writing to these registers enables specific interrupts.
- DICENABLERn - interrupt clear-enable registers. Like the above, but writing to these registers disables interrupts.
- DIPRIORITYRn - interrupt priorty registers. Lets each interrupt have a different priority level, with these priorities determining which interrupt actually gets forwarded to a CPU when there are multiple pending interrupts.
- DITARGETSRn - interrupt processor target registers. These determine which CPU will get notified for each interrupt.
- DICFGRn - interrupt configuration registers. They identify whether each interrupt is edge-triggered or level-sensitive. Edge-triggered interrupts can be deasserted (marked as no longer pending) by the peripheral that triggered them in the first place, level-sensitive interrupts can only be cleared by the CPU.

There are more Distributor registers but the ones above would let us get some interrupt handling in place. That's just the Distributor part of the GIC though, there's also the CPU interface part, with key registers including:

- CCTLR - CPU interface control register, enabling or disabling interrupt forwarding to the particular CPU connected to that interface.
- CCPMR - interrupt priority mask register. Acts as a filter of sorts between the Distributor and the CPUs - this register defines the minimum priority level for an intrrupt to be forwarded to the CPU.
- CIAR - interrupt acknowledge register. The CPU receiving the interrupt is expected to read from this register in order to obtain the interrupt ID, and thereby acknowledge the interrupt.

- CEOIR - end of interrupt register. The CPU is expected to write to this register after completing the handling of an interrupt.

First GIC implementation

Let us say that the first goal is to successfully react to an interrupt. For that, we will need a basic GIC driver and an interrupt handler, as well as some specific interrupt to enable and react to. The UART can act as an interrupt source, as a UART data reception (keypress in the terminal) triggers an interrupt. From there, we'll be able to iterate and improve the implementation with better interrupt hanlders and the use of vectorized interrupts.

This section has quite a lot of information and again refers to multiple manuals, so do not worry if it initially seems complicated!

We begin by defining the appropriate structures in a header file that could be called `gic.h`, taking the register map from the GIC manual as the source of information. The result looks something like this:

```
1  typedef volatile struct __attribute__((packed)) {
2      uint32_t DCTLR;                 /* 0x0 Distributor Control register
           */
3      const uint32_t DTYPER;          /* 0x4 Controller type register */
4      const uint32_t DIIDR;           /* 0x8 Implementer identification
           register */
5      uint32_t _reserved0[29];        /* 0xC - 0x80; reserved and
           implementation-defined */
6      uint32_t DIGROUPR[32];          /* 0x80 - 0xFC Interrupt group
           registers */
7      uint32_t DISENABLER[32];        /* 0x100 - 0x17C Interrupt set-
           enable registers */
8      uint32_t DICENABLER[32];        /* 0x180 - 0x1FC Interrupt clear-
           enable registers */
9      uint32_t DISPENDR[32];          /* 0x200 - 0x27C Interrupt set-
           pending registers */
10     uint32_t DICPENDR[32];          /* 0x280 - 0x2FC Interrupt clear-
           pending registers */
11     uint32_t DICDABR[32];           /* 0x300 - 0x3FC Active Bit
           Registers (GIC v1) */
12     uint32_t _reserved1[32];        /* 0x380 - 0x3FC reserved on GIC v1
           */
13     uint32_t DIPRIORITY[255];       /* 0x400 - 0x7F8 Interrupt priority
           registers */
14     uint32_t _reserved2;            /* 0x7FC reserved */
15     const uint32_t DITARGETSRO[8];  /* 0x800 - 0x81C Interrupt CPU
           targets, RO */
```

```
16      uint32_t DITARGETSR[246];       /* 0x820 - 0xBF8 Interrupt CPU
            targets */
17      uint32_t _reserved3;            /* 0xBFC reserved */
18      uint32_t DICFGR[64];            /* 0xC00 - 0xCFC Interrupt config
            registers */
19      /* Some PPI, SPI status registers and identification registers
            beyond this.
20         Don't care about them */
21  } gic_distributor_registers;
22
23  typedef volatile struct __attribute__((packed)) {
24      uint32_t CCTLR;                 /* 0x0 CPU Interface control
            register */
25      uint32_t CCPMR;                 /* 0x4 Interrupt priority mask
            register */
26      uint32_t CBPR;                  /* 0x8 Binary point register */
27      const uint32_t CIAR;            /* 0xC Interrupt acknowledge
            register */
28      uint32_t CEOIR;                 /* 0x10 End of interrupt register
            */
29      const uint32_t CRPR;            /* 0x14 Running priority register
            */
30      const uint32_t CHPPIR;          /* 0x18 Higher priority pending
            interrupt register */
31      uint32_t CABPR;                 /* 0x1C Aliased binary point
            register */
32      const uint32_t CAIAR;           /* 0x20 Aliased interrupt
            acknowledge register */
33      uint32_t CAEOIR;                /* 0x24 Aliased end of interrupt
            register */
34      const uint32_t CAHPPIR;         /* 0x28 Aliased highest priority
            pending interrupt register */
35  } gic_cpu_interface_registers;
```

There is nothing particularly noteworthy about the structs, they follow the same patterns as explained in the previous chapter. Note that Distributor and CPU Interface stuctures cannot be joined together because they may not be contiguous in memory (and indeed aren't on the Cortex-A CPUs).

When that's done, we need to write `gic.c`, our implementation file. The first version can be really simple, but it will nonetheless reveal several things that we had not had to consider before. JHere's how `gic.c` begins:

```
1  #include "gic.h"
2
3  static gic_distributor_registers* gic_dregs;
4  static gic_cpu_interface_registers* gic_ifregs;
5
6  void gic_init(void) {
```

```
7       gic_ifregs = (gic_cpu_interface_registers*)GIC_IFACE_BASE;
8       gic_dregs = (gic_distributor_registers*)GIC_DIST_BASE;
9
10      WRITE32(gic_ifregs->CCPMR, 0xFFFFu); /* Enable all interrupt
            priorities */
11      WRITE32(gic_ifregs->CCTLR, CCTRL_ENABLE); /* Enable interrupt
            forwarding to this CPU */
12
13      gic_distributor_registers* gic_dregs = (gic_distributor_registers*)
            GIC_DIST_BASE;
14      WRITE32(gic_dregs->DCTLR, DCTRL_ENABLE); /* Enable the interrupt
            distributor */
15  }
```

We define static variables to hold pointers to the Distributor and the CPU Interface, and write an initialization function. Here you might already notice one difference from the UART driver earlier. The UART driver had its pointer initialized to the hardware address the hardware uses, like this:

```
1  static uart_registers* uart0 = (uart_registers*)0x10009000u;
```

With GIC registers, we cannot do the same because their address is implementation-dependent. Hardcoding the address for a particular board is possible (and it is what QEMU itself does) but we can implement the more correct way, setting those register addresses in `gic_init`. The Cortex-A9 MPCore manual states that the GIC is within the CPU's private memory region, specifically the CPU interface is at `0x0100` from `PERIPHBASE` and the Distributor is at `0x1000` from `PERIPHBASE`. What's this `PERIPHBASE` then? The A9 MPCore manual also states that:

PERIPHBASE[31:13]	I	Specifies the base address for Timers, Watchdogs, Interrupt Controller, and SCU registers. Only accessible with memory-mapped accesses. This value can be retrieved by a Cortex-A9 processor using the CP15 c15 Configuration Base Address Register.

Figure 7.1: Description of PERIPHBASE

It should be clear that the GIC Distributor is located at `PERIPHBASE + 0x1000` but obtaining `PERIPHBASE` seems confusing. Let's take a look at the `GIC_DIST_BASE` and `GIC_IFACE_BASE` macros that `gic_init` uses.

```
1  #define GIC_DIST_BASE    ((cpu_get_periphbase() + GIC_DISTRIBUTOR_OFFSET
       ))
2  #define GIC_IFACE_BASE   ((cpu_get_periphbase() + GIC_IFACE_OFFSET))
```

I put the offsets themselves into a different CPU-specific header file `cpu_a9.h`, but it can of course be organized however you want. The `cpu_get_periphbase` function is implemented like this:

```
1 inline uint32_t cpu_get_periphbase(void) {
2     uint32_t result;
3     asm ("mrc p15, #4, %0, c15, c0, #0" : "=r" (result));
4     return result;
5 }
```

Just what is going on there? It's a function with a weirdly-formatted assembly line, and the assembly itself refers to strange things like `p15`. Let's break this down.

C functions can use what is known as *inline assembly* in order to include assembly code directly. Inline assembly is generally used either for manual optimization of critical code, or to perform operations that are not exposed to ordinary code. We have the latter case. When writing inline assembly for GCC, you can use the *extended assembly syntax*, letting you read or write C variables. When you see a colon `:` in an inline assembly block, that's extended assembly syntax, which is documented by GCC and in the simplest case looks like `asm("asm-code-here": "output")`, where the `output` refers to C variables that will be modified.

The `%0` part in our extended assembly block is just a placeholder, and will be replaced by the first (and, in this case, the only) output operand, which is `"=r"(result)`. That output syntax in turn means that we want to use some register (`=r`) and that it should write to the `result` variable. The choice of the specific register is left to GCC. If we were writing in pure assembly, the instruction would be, assuming the `R0` register gets used for output

```
1 mrc p15, #4, r0, c15, c0, #0
```

That's still one strange-looking instruction. ARM processors (not just ARMv7 but also older architectures) support *coprocessors*, which may include additional functionality outside the core processor chip itself. Coprocessor 15, or CP15 for short, is dedicated to important control and configuration functions. Coprocessors are accessed through the special instructions `mrc` (read) and `mcr` (write). Those instructions contain additional opcodes, the meaning of which depends on the coprocessor.

The A9 MPCore manual makes a reference to the "CP15 c15 Configuration Base Address Register" when describing `PERIPHBASE`. CP15 is, as we now know, coprocessor 15, but the `c15` part refers, confusingly, to something else, namely to a specific register in CP15. The `mrc` instruction has a generic format, which is:

```
1 mrc coproc, op1, Rd, CRn, CRm [,op2]
```

So the coprocessor number comes first, `Rd` refers to the ARM register to read data to, while `op1` and optionally `op2` are operation codes defined by the coprocessor, and `CRn` and `CRm` are coprocessor registers. This means that, in order to do something with a coprocessor, we need to look up its own documentation. The coprocessor's features fall under the corresponding processor features, and we can find what interests us in the Cortex A9 manual. Chapter 4, *System Control* concerns CP15, and a register summary lists the various operations and registers that are available. Under c15, we find the following:

4.2.16 c15 registers

Table 4-15 shows the CP15 system control registers you can access when CRn is c15.

Table 4-15 c15 system control register summary

Op1	CRm	Op2	Name	Type	Reset	Description
0	c0	0	Power Control Register	RW[a]	- [b]	*Power Control Register* on page 4-41
	c1	0	NEON Busy Register	RO	0x00000000	*NEON Busy Register* on page 4-42
4	c0	0	Configuration Base Address	RO[c]	- [d]	*Configuration Base Address Register* on page 4-42
5	c4	2	Select Lockdown TLB Entry for read	WO[e]	-	*TLB lockdown operations* on page 4-43
		4	Select Lockdown TLB Entry for write	WO[e]	-	
	c5	2	Main TLB VA register	RW[e]	-	
	c6	2	Main TLB PA register	RW[e]	-	
	c7	2	Main TLB Attribute register	RW[e]	-	

Figure 7.2: CP15 c15 register summary

Looking through the table, we can finally find out that reading the Configuration Base Register, which contains the `PERIPHBASE` value, requires accessing CP15 with `Rn=c15`, `op1 = 4`, `CRm = c0`, and `op2 = 0`. Putting it all together gives the `mrc` instruction that we use in `cpu_get_periphbase`.

The remainder of `gic_init` is quite unremarkable. We enable forwarding of interrupts with all priorities to the current CPU, and enable the GIC Distributor so that interrupts from external sources could reach the CPU. Note the use of the `WRITE32` macro. Register access width was mentioned in the previous chapter, and unlike the PL011 UART, the GIC explicitly states that all registers permit 32-bit word access, with only a few Distributor registers allowing byte access. So we should take care to write the registers with one 32-bit write with this macro.

```
#define WRITE32(_reg, _val) (*(volatile uint32_t*)&_reg = _val)
```

The next order of business is to let specific interrupts be enabled. Initializing the GIC means we can now receive interrupts in general. As said before, upon receiving an interrupt, the GIC Distributor checks if the particular interrupt is enabled before forwarding it to the CPU interface. The Set-Enable registers, GICD_ISENABLER[n], control whether a particular interrupt is enabled. Each ISENABLER register can enable up to 32 interrupts, and having many such registers allows the hardware to have more than 32 different interrupt sources. Given an interrupt with id `N`, enabling it means setting the bit `N % 32` in register `N / 32`, where integer division is used. For example, interrupt 45 would be bit 13 (`45 % 32 = 13`) in ISENABLER[1] (`45 / 32 = 1`).

For each interrupt, you also need to select which CPU interface(s) to forward the interrupt to, done in the GICD_ITARGETSR[n] registers. The calculation for these registers is slightly different, for interrupt with id `N` the register is `N / 4`, and the target list has to be written to byte `N % 4` in that register. The target list is just a byte where bit 0 represents CPU interface 0, bit 1 represents CPU interface 1 and so on. We don't need anything fancy here, we just want to forward any enabled interrupts to CPU Interface 0.

With that knowledge, writing the following function becomes quite simple:

```
void gic_enable_interrupt(uint8_t number) {
    /* Enable the interrupt */
    uint8_t reg = number / 32;
    uint8_t bit = number % 32;

    uint32_t reg_val = gic_dregs->DISENABLER[reg];
    reg_val |= (1u << bit);
    WRITE32(gic_dregs->DISENABLER[reg], reg_val);

    /* Forward interrupt to CPU Interface 0 */
    reg = number / 4;
    bit = (number % 4) * 8; /* Can result in bit 0, 8, 16 or 24 */
    reg_val = gic_dregs->DITARGETSR[reg];
    reg_val |= (1u << bit);
    WRITE32(gic_dregs->DITARGETSR[reg], reg_val);
}
```

Now we have `gic_init` to initialize the GIC and `gic_enable_interrupt` to enable a specific interrupt. The preparation is almost done, we just need functions to globally disable and enable interrupts. When using an interrupt controller, it's a good idea to disable interrupts on startup, and then enable them after the interrupt controller is ready.

Disabling interrupts is easy, we can do it somewhere in the assembly startup code in `startup.s`. At some point when the CPU is in supervisor mode, add the `cpsid if` instruction to disable all interrupts

- the `if` part means both IRQs and FIQs. One possible place to do that would be right before the `bl main` instruction that jumps to C code.

Enabling interrupts is done similarly, with the `cpsie if` instruction. We'll want to call this from C code eventually so it's convenient to create a C function with inline assembly in some header file, like this:

```
1 inline void cpu_enable_interrupts(void) {
2     asm ("cpsie if");
3 }
```

Looks like we're done! Just to make sure the new functions are getting used, call `gic_init()` and then `cpu_enable_interrupts()` from somewhere in the `main` function (after the initial UART outputs perhaps). At this point you can try building the program (remember to add `gic.c` to the source file list in `CMakeLists.txt`), but surprisingly enough, the program won't compile, and you'll get an error like

```
1 /tmp/ccluurNJ.s:146: Error: selected processor does not support  cpsie
     if' in ARM mode
```

This is our first practical encounter with the fact that ARMv7 (same goes for some other ARM architectures) has two instruction sets, ARM and Thumb (Thumb version 2 to be exact). Thumb instructions are smaller at 16 bits compared to the 32 bits of an ARM instruction, and so can be more efficient, at the cost of losing some flexibility. ARM CPUs can freely switch between the two instruction sets, but Thumb should be the primary choice. In the above error message, GCC is telling us that `cpsie if` is not available as an ARM instruction. It is indeed not, it's a Thumb instruction. We need to change the compiler options and add `-mthumb` to request generation of Thumb code. In `CMakeLists.txt`, that means editing the line that sets `CMAKE_C_FLAGS`. After adding `-mthumb` to it we can try to recompile. The interrupt-enabling instruction no longer causes any problems but another issue crops up:

```
1 /tmp/ccC72j7I.s:37: Error: selected processor does not support  mrc p15
     ,#4,r2,c15,c0,#0' in Thumb mode
```

Indeed, accessing the coprocessors is only possible with ARM instructions. The `mrc` instruction does not exist in the Thumb instruction set. It's possible to control the CPU's instruction set and freely switch between the two, but fortunately, GCC can figure things out by itself if we tell it what specific CPU we're using. So far we've just been compiling for a generic ARM CPU, but we can easily specify the CPU by also adding `-mcpu=cortex-a9` to the compilation flags. So now with `-mthumb -mcpu=cortex-a9` added to the compile flags, we can finally compile and run the application just as before.

You should see that the program works just like it did at the end of the previous chapter. Indeed, we've enabled the GIC, and have interrupts enabled globally for the CPU, but we haven't enabled any specific interrupts yet, so the GIC will never forward any interrupts that may get triggered.

NOTE

With the GIC enabled, you can view its registers with a debugger or in the QEMU monitor, with some caveats. If you're using QEMU older than version 3.0, then the Distributor's control register will show the value `0` when read that way, even if the Distributor is actually enabled. And if you try to access the CPU Interface registers (starting at `0x1e000100`) with an external debugger like GDB, QEMU will crash, at least up to and including version 3.1.0

Handling an interrupt

Let's now put the GIC to use and enable the UART interrupt, which should be triggered any time the UART receives data, which corresponds to us pressing a key in the terminal when running with QEMU. After receiving an interrupt, we'll need to properly handle it to continue program execution.

Enabling the UART interrupt should be easy since we already wrote the `gic_enable_interrupt` function, all we need to do now is to call it with the correct interrupt number. That means once again going back to the manuals to find the interrupt ID numbe we need to use. Interrupt numbers usually differ depending on the board, and in our case the CoreTile Express A9x4 manual can be the first stop. The section *2.6 Interrupts* explains that the integrated test chip for the Cortex-A9 MPCore on this board is directly connected to the motherboard (where the UART is located as we remember from the previous chapter), and that motherboard interrupts 0-42 map to interrupts 32-74 on the daughterboard. This means we need to check the motherboard manual and add 32 to the interrupt number we find there.

The Motherboard Express µATX manual explains in *2.6 Interrupt signals* that the motherboard has no interrupt controller, but connects interrupt signals to the daughterboard. The signal list says that `UART0INTR`, the interrupt signal for UART0, is number `5`. Since the daughterboard remaps interrupts, we'll need to enable interrupt `37` in order to receive UART interrupts in our program. The following snippet in `main` should do just fine:

```
gic_init();
gic_enable_interrupt(37);
cpu_enable_interrupts();
```

And we need an interrupt handler, which we need to point out in the vector table in `startup.s`. It should now look something like

```
_Reset:
    b Reset_Handler
    b Abort_Exception /* 0x4  Undefined Instruction */
    b . /* 0x8  Software Interrupt */
    b Abort_Exception  /* 0xC  Prefetch Abort */
    b Abort_Exception /* 0x10 Data Abort */
    b . /* 0x14 Reserved */
    b IrqHandler /* 0x18 IRQ */
    b . /* 0x1C FIQ */
```

The seventh entry in the vector table, at offset `0x18`, will now jump to `IrqHandler`. We can add it to the end of `startup.s`, and the simplest implementation that would tell us things are working fine can just store the data that the UART received in some register and hang.

```
IrqHandler:
    ldr r0, =0x10009000
    ldr r1, [r0]
    b .
```

Reading from the UART register at `0x10009000` gives the data byte that was received, and we proceed to store it in `R1` before hanging. Why hang? Continuing execution isn't as simple as just returning from the IRQ handler, you have to take care to save the program state before the IRQ, then restore it, which we're not doing. Our handler, the way it's written above, breaks the program state completely.

Let's compile and test now! Once the program has started in QEMU and written its greetings to the UART, press a key in the terminal to trigger the now-enabled UART interrupt. Then check the registers with `info registers` in QEMU monitors, and unfortunately you'll notice a problem. The IRQ handler doesn't seem to be running and the program is just hanging. Output could be something similar to:

```
(qemu) info registers
R00=00000005 R01=00000000 R02=00000008 R03=00000090
R04=00000000 R05=7ffd864c R06=60000000 R07=00000000
R08=00000400 R09=7fef5ef8 R10=00000001 R11=00000001
R12=00000000 R13=00000013 R14=7ff96264 R15=7ff96240
PSR=00000192 ---- A S irq32
```

Good news first, the program status register `PSR` indicates that the CPU is running in IRQ mode (`0x192 & 0x1F` is `0x12`, which is IRQ mode, but QEMU helpfully points it out by writing `irq32` on the same line). The bad news is that `R0` and `R1` don't contain the values we would expect from `IrqHandler`, and the CPU seems to be currently running code at some strange address in `R15` (remember that `R15` is just another name for `PC`, the program counter register). The address doesn't correspond to anything we've loaded into memory so the conclusion is that the CPU did receive an interrupt, but failed to run `IrqHandler`.

This is one more detail that happens due to QEMU emulation not being perfect. If you remember the discussion about memory and section layout from Chapter 4, we're pretending that our ROM starts at `0x60000000`. The Cortex-A9 CPU, however, expects the vector table to be located at address `0x0`, according to the architecture, and IRQ handling starts by executing the instruction at `0x18` from the vector table base. Unfortunately, our vector table is actually at `0x60000000` and address `0x0` is reserved by QEMU for the program flash memory, which we cannot emulate.

We then need to make a QEMU-specific modification to our code and indicate that the vector table base is at `0x60000000`. This is a very low-level modification of the CPU configuration, so you might be able to guess that the system control coprocessor, CP15, is involved again. We previously used its `c15` register to read `PERIPHBASE`, and the ARMv7-A manual will reveal that the `c12` register contains the vector table base address, which may also be modified. To write to the coprocessor, we use the `mcr` instruction (as opposed to `mrc` for reading), and the instructions we need will be:

```
1 ldr r0, =0x60000000
2 mcr p15, #0, r0, c12, c0, #0
```

Those two instructions should be somewhere early in the startup code, such as right after the `Reset_Handler` label. Having done that modification, we can perform another rebuild and test run. Press a key in the terminal, and check the registers in the QEMU monitor. Now you should see that `R0` contains the UART address, and `R1` contains the code code of the key you pressed, such as `0x66` for `f` or `0x61` for `a`.

```
1 R00=10009000 R01=00000066 R02=00000008 R03=00000090
```

With that, we have correctly jumped into an interrupt handler after an external interrupt triggers, which is a major step towards improving our bare-metal system.

Surviving the IRQ handler

Our basic implementation of the IRQ handler isn't good for much, the biggest issue being that the program hangs completely and never leaves the IRQ mode.

Interrupt handlers, as the name suggests, interrupt whatever the program was doing previously. This means that state needs to be saved before the handler, and restored after. The general-purpose ARM registers, for example, are shared between modes, so if your register `R0` contains something, and then an interrupt handler writes to it, the old value is lost. This is part of the reason why a separate IRQ stack is needed (which we prepare in the startup code), as the IRQ stack is normally where the context would be saved.

When writing interrupt handlers in assembly, we have to take care of context saving and restoring, and correctly returning from the handler. Hand-written assembly interrupt handlers should be reserved for cases where fine-tuned assembly is critical, but generally it's much easier to write interrupt handlers in C, where they become regular functions for the most part. The compiler can handle context save and restore, and everything else that's needed for interrupt handling, if told that a particular function is an interrupt handler. In GCC, `__attribute__((interrupt))` is a decoration that can be used to indicate that a function is an interrupt handler.

We can write a new function in the UART driver that would respond to the interrupt.

```
1 void __attribute__((interrupt)) uart_isr(void) {
2     uart_write("Interrupt!\n");
3 }
```

Then just changing `b IrqHandler` to `b uart_isr` in the vector table will ensure that the `uart_isr` function is the one called when interrupts occur. If you test this, you'll see that the program just keeps spamming `Interrupt!` endlessly after a keypress. Our ISR needs to communicate with the GIC, acknowledge the interrupt and signal the GIC when the ISR is done. In the GIC, we need a function that acknowledges an interrupt.

```
1 uint32_t gic_acknowledge_interrupt(void) {
2     return gic_ifregs->CIAR & CIAR_ID_MASK;
3 }
```

`CIAR_ID_MASK` is `0x3FF` because the lowest 9 bits of `CIAR` contain the interrupt ID of the interrupt that the GIC is signaling. After a read from `CIAR`, the interrupt is said to change from pending state to active. Another function is necessary to signal the end of the interrupt, which is done by writing the interrupt ID to the `EOIR` register.

```
void gic_end_interrupt(uint16_t number) {
    WRITE32(gic_ifregs->CEOIR, (number & CEOIR_ID_MASK));
}
```

The ISR could then use those two functions and do something along the lines of:

```
void __attribute__((interrupt)) uart_isr(void) {
    uint16_t irq = gic_acknowledge_interrupt();
    if (irq == 37) {
        uart_write("Interrupt!\n");
    }
    gic_end_interrupt(37);
}
```

This implementation is better but would still result in endless outputs. The end-of-interrupt would be correctly signaled to the GIC, but the GIC would forward a new UART interrupt to the CPU. This is because the interrupt is generated by the UART peripheral, the GIC just forwards it. The code above lets the GIC know we're done handling the interrupt, but doesn't inform the UART peripheral of that. The PL011 UART has an interrupt clear register, ICR, which is already in the header file from the last chapter. Clearing all interrupts can be done by writing `1` to bits `0-10`, meaning the mask is `0x7FF`. If we clear all interrupt sources in the UART before signaling end-of-interrupt to the GIC, everything will work.

```
void __attribute__((interrupt)) uart_isr(void) {
    uint16_t irq = gic_acknowledge_interrupt();
    if (irq == 37) {
        uart_write("Interrupt!\n");
    }
    uart0->ICR = ICR_ALL_MASK;
    gic_end_interrupt(37);
}
```

With that interrupt handler, our program will write `Interrupt!` every time you press a key in the terminal, after which it will resume normal execution. You can verify for yourself that the CPU returns to the supervisor (SVC) mode after handling the interrupt. It can also be interesting to disassemble the ELF file and note how the code for `uart_isr` differs from any other functions - GCC will have generated `stmdb` and `ldmia` instructions to save several registers to the stack and restore them later.

Adapting the UART driver

We now finally have working interrupt handling with a properly functional ISR that handles an interrupt, clears the interrupt source and passes control back to whatever code was running before the interrupt. Next let us apply interrupts in a useful manner, by adapting the UART driver and making the interrupts do something useful.

The first thing to note is that what we've been calling "the UART interrupt" is a specific interrupt signal, `UART0INT` that the motherboard forwards to the GIC. From the point of view of the PL011 UART itself though, several different interrupts exist. The PL011 manual has a section devoted to interrupts, which lists eleven different interrupts that the peripheral can generate, and it also generates an interrupt `UARTINTR` that is an OR of the individual interrupts (that is, `UARTINTR` is active if any of the others is). It's this `UARTINTR` that corresponds to the interrupt number 37 which we enabled, but our driver code should check which interrupt occurred specifically and react accordingly.

The `UARTMIS` register can be used to read the masked interrupt status, with the `UARTRIS` providing the raw interrupt status. The difference between those is that, if an interrupt is masked (disabled) in the UART's configuration, it can still show as active in the raw register but not the masked one. By default all interrupts all unmasked (enabled) on the PL011 so this distinction doesn't matter for us. Of the individual UART interrupts, only the receive interrupt is really interesting in the basic use case, so let's implement that one properly, as well as one of the error interrupts.

All interrupt-related PL011 registers use the same pattern, where bits 0-10 correspond to the same interrupts. The receive (RX) interrupt is bit 4, the break error (BE) interrupt is bit 9. We can express that nicely with a couple of defines:

```
#define RX_INTERRUPT    (1u << 4u)
#define BE_INTERRUPT    (1u << 9u)
```

We're using the UART as a terminal, so when the receive interrupt occurs, we'd like to print the character that was received. If the break error occurs, we can't do much except clear the error flag (in the RSRECR register) and write an error message. Let's write a new ISR that checks for the actual underlying UART interrupt and reacts accordingly.

```
void __attribute__((interrupt)) uart_isr(void) {
    (void)gic_acknowledge_interrupt();

    uint32_t status = uart0->MIS;
    if (status & RX_INTERRUPT) {
        /* Read the received character and print it back*/
```

```
        char c = uart0->DR & DR_DATA_MASK;
        uart_putchar(c);
        if (c == '\r') {
            uart_write("\n");
        }
    } else if (status & BE_INTERRUPT) {
        uart_write("Break error detected!\n");
        /* Clear the error flag */
        uart0->RSRECR = ECR_BE;
        /* Clear the interrupt */
        uart0->ICR = BE_INTERRUPT;
    }

    gic_end_interrupt(UART0_INTERRUPT);
}
```

In the previous chapter, we had a loop in `main` that polled the UART. That is no longer necessary, but remember that `main` should not terminate so the `while (1)` loop should still be there. The terminal functionality is now available and interrupt-driven!

Handling different interrupt sources

The interrupt handling solution at this point has a major flaw. No matter what interrupt the CPU receives, the `b uart_isr` from the vector table will take us to that interrupt handler, which is of course only suitable for the UART interrupt. Early on in this chapter, there was mention of vectored interrupts, which we cannot use since our hardware uses the GIC, a non-vectored interrupt controller. Therefore we'll need to use a software solution, writing a top-level interrupt handler that will be responsible for finding out which interrupt got triggered and then calling the appropriate function.

In the simplest case, we'd then write a function like the following:

```
void __attribute__((interrupt)) irq_handler(void) {
        uint16_t irq = gic_acknowledge_interrupt();
        switch (irq) {
        case UART0_INTERRUPT:
            uart_isr();
            break;
        default:
            uart_write("Unknown interrupt!\n");
            break;
        }
        gic_end_interrupt(irq);
}
```

This top-level `irq_handler` should then be pointed to by the vector table, and adding support for new interrupts would just mean adding them to the **`switch`** statement. The top-level handler takes care of the GIC acknowledge/end-of-interrupt calls, so individual handlers like `uart_isr` no longer have to do it, nor do they need the `__attribute__((interrupt))` anymore because the top-level handler is where the switch to IRQ mode should happen.

Purely from an embedded code perspective, there's no problem with such a handler and having a long list of interrupts in the **`switch`** statement. It's not a great solution in terms of general software design though. It creates quite tight coupling between the top-level IRQ handler, which should be considered to be a separate module from the GIC, and the handler would have to know about all other relevant modules. If we place the above handler into a separate file like `irq.c`, it would have to include `uart_pl011.h` for the header's declaration of `uart_isr`. If we then add a timer module, `irq.c` would also need to include `timer.h` and `irq_handler` would have to be modified to call some timer ISR, which is not a good, maintainable way to structure the code.

A better solution is to make the IRQ handler use callbacks, and allow individual modules to register those callbacks. We can then offload some important work to a separate IRQ component, with `irq.h`:

```
#ifndef IRQ_H
#define IRQ_H

#include <stdint.h>

typedef void (*isr_ptr)(void);

#define ISR_COUNT   (1024)
#define MAX_ISR     (ISR_COUNT - 1)


typedef enum {
    IRQ_OK = 0,
    IRQ_INVALID_IRQ_ID,
    IRQ_ALREADY_REGISTERED
} irq_error;

irq_error irq_register_isr(uint16_t irq_number, isr_ptr callback);

#endif
```

The header defines a function `irq_register_isr` that other modules would then call to register their own ISRs. The `isr_ptr` type is a function pointer to an ISR - `typedef void (*isr_ptr)(void);` means that `isr_ptr` is a pointer to a function that returns **`void`** and takes no parameters. If the syntax is confusing, take a moment to read up on C function pointers online - conceptually function pointers are not difficult but the syntax tends to feel obscure until you get used to it.

The implementation in `irq.c` is:

```
#include <stddef.h>
#include "irq.h"
#include "gic.h"

static isr_ptr callbacks[1024] = { NULL };

static isr_ptr callback(uint16_t number);

void __attribute__((interrupt)) irq_handler(void) {
    uint16_t irq = gic_acknowledge_interrupt();
    isr_ptr isr = callback(irq);
    if (isr != NULL) {
        isr();
    }
    gic_end_interrupt(irq);
}

irq_error irq_register_isr(uint16_t irq_number, isr_ptr callback) {
    if (irq_number > MAX_ISR) {
        return IRQ_INVALID_IRQ_ID;
    } else if (callbacks[irq_number] != NULL) {
        return IRQ_ALREADY_REGISTERED;
    } else {
        callbacks[irq_number] = callback;
    }
    return IRQ_OK;
}

static isr_ptr callback(uint16_t number) {
    if (number > MAX_ISR) {
        return NULL;
    }
    return callbacks[number];
}
```

We use an array that can store up 1024 ISRs, which is enough to use all the interrupts the GIC supports if desired. The top-level `irq_handler` talks to the GIC and calls whatever ISR has been registered for the particular interrupt. The UART driver then registers its own ISR in `uart_init` just before enabling the UART peripheral, like this:

```
/* Register the interrupt */
(void)irq_register_isr(UART0_INTERRUPT, uart_isr);
```

Such a solution no longer requires the IRQ handler to know which specific ISRs exist beforehand, and

is easier to maintain.

Summary

In this chapter, we went over interrupt handling in general, the ARM Generic Interrupt Controller, and we wrote some interrupt handlers.

Interrupts are often among the trickiest topics in embedded development. Interrupt controllers themselves are quite complicated - we used the GIC in pretty much the simplest way possible, but it can quickly get complicated once you start grouping interrupts, working with their priorities and so on. Another complication arises from the hard-to-predict nature of interrupts. You don't know what regular code will be executed when an interrupt happens. Many interrupts in the real world depend on timing or external data sources, so debugging with breakpoints affects the behavior of the program.

As a broad generalization, interrupt handling becomes trickier and more important as you develop on more limited hardware. When dealing with microcontrollers, you often have to understand the amount of time spent in interrupts, and may also find that the switching between normal and IRQ modes creates real performance issues. Fast interrupts, FIQs, which we didn't cover in this chapter exist in ARMv7 to help alleviate the overhead of regular IRQs.

In a real embedded system that does something useful, interrupts are likely to drive some critical parts of functionality. For example, most systems need some way of measuring time or triggering some code in a time-based manner, and that usually happens by having a timer that generates interrupts.

8 Simple scheduling

Very few embedded applications can be useful without some kind of time-keeping and scheduling. Being able to program things like “do this every X seconds” or “measure the time between events A and B” is a key aspect of nearly all practically useful applications.

In this chapter, we’ll look at two related concepts. *Timers*, which are a hardware feature that allows software to keep track of time, and *scheduling*, which is how you program a system to run some code, or a *task*, on some kind of time-based schedule - hence the name. Scheduling in particular is a complex subject, some discussion of which will follow later, but first we’ll need to set up some kind of time measurement system.

In order to keep track of time in our system, we’re going to use two tiers of “ticks”. First, closer to the hardware, we’ll have a timer driver for a hardware timer of the Cortex-A9 CPU. This driver will generate interrupts at regular intervals. We will use those interrupts to keep track of *system time*, a separate counter that we’ll use in the rest of the system as “the time”.

Such a split is not necessary in a simple tutorial system, but is good practice due to the system time thus not being directly connected to a particular hardware clock or driver implementation. This allows better portability as it becomes possible to switch the underlying timer driver without affecting uses of system time.

The first task then is to create a timer driver. Since its purpose will be to generate regular interrupts, note that this work builds directly on the previous chapter, where interrupt handling capability was added.

Private Timer Driver

A Cortex-A9 MPCore CPU provides a global timer and private timers. There’s one private timer per core. The global timer is constantly counting up, even with the CPU paused in debug mode. The per-core private timers count down from some starting value to zero, sending an interrupt when zero is reached. It’s possible to use either timer for scheduling, but the typical solution is to use the private timer. It’s somewhat easier to handle due to being 32 bits wide (the global timer is 64 bits) and due to stopping when the CPU is stopped.

The Cortex-A9 MPCore manual explains the private timer in Chapter 4.1. The timer's operation is quite simple. A certain starting *load value* is loaded into the load register `LR`. When the timer is started, it keeps counting down from that load value, and generates an interrupt when the counter reaches `0`. Then, if the auto-reload function is enabled, the counter automatically restarts from the load value. As is common with other devices, the private timer has its own control register `CTRL`, which controls whether auto-reload is enabled, whether interrupt generation is enabled, and whether the whole timer is enabled.

From the same manual, we can see that the private timer's registers are at offset `0x600` from `PERIPHBASE`, and we already used `PERIPHBASE` in the previous chapter for GIC registers. Finally, the manual gives the formula to calculate the timer's period.

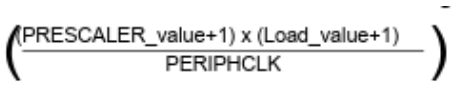


Figure 8.1: Private timer period formula

A prescaler can essentially reduce the incoming clock frequency, but using that is optional. If we simplify with the assumption that prescaler is `0`, we can get `Load value = (period * PERIPHCLK)- 1`. The peripheral clock, `PERIPHCLK`, is the same 24 MHz clock from the motherboard that clocks the UART.

I will not go through the timer driver in every detail here, as it just applies concepts from the previous two chapters. As always, you can examine the full source in this chapter's corresponding source code folder. We call the driver `ptimer`, implement it in `ptimer.h` and `ptimer.c`, and the initialization function is as follows:

```
ptimer_error ptimer_init(uint16_t millisecs) {
    regs = (private_timer_registers*)PTIMER_BASE;
    if (!validate_config(millisecs)) {
        return PTIMER_INVALID_PERIOD;
    }
    uint32_t load_val = millisecs_to_timer_value(millisecs);
    WRITE32(regs->LR, load_val); /* Load the initial timer value */

    /* Register the interrupt */
    (void)irq_register_isr(PTIMER_INTERRUPT, ptimer_isr);

    uint32_t ctrl = CTRL_AUTORELOAD | CTRL_IRQ_ENABLE | CTRL_ENABLE;
    WRITE32(regs->CTRL, ctrl); /* Configure and start the timer */

    return PTIMER_OK;
}
```

The function accepts the desired timer tick period, in milliseconds, calculates the corresponding load value for `LR` and then enables interrupt generation, auto-reload and starts the timer.

One bit of interest here is the ISR registration, for which the interrupt number is defined as `#define PTIMER_INTERRUPT (29u)`. The CPU manual says that the private timer generates interrupt `29` when the counter reaches zero. And in the code we're actually using `29`, unlike with the UART driver, where we mapped `UART0INTR`, number `5`, to `37` in the code. But this interrupt remapping only applies to certain interrupts that are generated on the motherboard. The private timer is part of the A9 CPU itself, so no remapping is needed.

The `millisecs_to_timer_value` function calculates the value to be written into `LR` from the desired timer frequency in milliseconds. Normally it should look like this:

```
static uint32_t millisecs_to_timer_value(uint16_t millisecs) {
    double period = millisecs * 0.001;
    return (period * refclock) - 1;
}
```

However, things are quite different for us due to using QEMU. It doesn't emulate the 24 MHz peripheral clock, and QEMU in general does not attempt to provide timings that are similar to the hardware being emulated. For our UART driver, this means that the baud rate settings don't have any real effect, but that wasn't a problem. For the timer though, it means that the period won't be the same as on real hardware, so the actual implementation used for this tutorial is:

```
static uint32_t millisecs_to_timer_value(uint16_t millisecs) {
    double period = millisecs * 0.001;
    uint32_t value = (period * refclock) - 1;
    value *= 3; /* Additional QEMU slowdown factor */

    return value;
}
```

With the timer driver in place, the simplest way to test is to make the timer ISR print something out, and initialize the timer with a one-second period. Here's the straightforward ISR:

```
void ptimer_isr(void) {
    uart_write("Ptimer!\n");
    WRITE32(regs->ISR, ISR_CLEAR); /* Clear the interrupt */
}
```

And somewhere in our `main` function:

```
gic_enable_interrupt(PTIMER_INTERRUPT);

if (ptimer_init(1000u) != PTIMER_OK) {
    uart_write("Failed to initialize CPU timer!\n");
}
```

Note the call to `gic_enable_interrupt`, and recall that each interrupt needs to be enabled in the GIC, it's not enough to just register a handler with `irq_register_isr`. This code should result in the private timer printing out a message every second or, due to the very approximate calculation in the emulated version, approximately every second.

Build everything and run (if you're implementing yourself as you read, remember to add the new C file to `CMakeLists.txt`), and you should see regular outputs from the timer.

HINT

You can use `gawk`, the GNU version of `awk` to print timestamps to the terminal. Instead of just `make run`, type `make run | gawk '{ print strftime("[%H:%M:%S]"), $0 }'` and you'll see the local time before every line of output. This is useful to ascertain that the private timer, when set to `1000` milliseconds, procudes output roughly every second.

System Time

As discussed previously, we want to use some kind of *system time* system-wide. This is going to have a very straightforward implementation. The private timer will tick every millisecond, and its ISR will increment the system time. So system time itself will also be measured in milliseconds. Then `systime.c` is exceedingly simple:

```
#include "systime.h"

static volatile systime_t systime;

void systime_tick(void) {
    systime++;
}
```

```
8
9  systime_t systime_get(void) {
10     return systime;
11 }
```

The `systime_t` type is defined in the corresponding header file, as `typedef uint32_t systime_t`.

To make use of this, the private timer's ISR is modified so it simply calls `systime_tick` after clearing the hardware interrupt flag.

```
1  void ptimer_isr(void) {
2      WRITE32(regs->ISR, ISR_CLEAR); /* Clear the interrupt */
3      systime_tick();
4  }
```

That's it, just change the `ptimer_init` call in `main` to use a 1-millisecond period, and you have a system-wide time that can be accessed whenever needed.

Overflows and spaceships

A discussion of timers is an opportune time to not only make a bad pun but also to mention overflows. By no means limited to embedded programming, overflows are nonetheless more prominent in low-level systems programming. As a refresher, an overflow occurs when the result of a calculation exceeds the maximum range of its datatype. A `uint8_t` has the maximum value `255` and so `255 + 1` would cause an overflow.

Timers in particular tend to overflow. For example, our use of `uint32_t` for system time means that the maximum system timer value is `0xFF FF FF FF`, or just shy of 4.3 billion in decimal. A timer that ticks every millisecond will reach that number after 49 days. So code that assumes a timer will always keep inceasing can break in mysterious ways after 49 days. This kind of bug is notoriously difficult to track to down.

One solution is of course to use a bigger data type. Using a 64-bit integer to represent a millisecond timer would be sufficient for 292 billion years. This does little to address problems in older systems, however. Many UNIX-based systems begin counting time from the 1st of January, 1970, and use a 32-bit integer, giving rise to what's known as the Year 2038 problem, as such systems cannot represent any time after January 19, 2038.

When overflows are possible, code should account for them. Sometimes overflows can be disregarded, but saying that something "cannot happen" is dangerous. It's reasonable to assume, for example, that

a microwave oven won't be running for 49 days in a row, but in some circumstances such assumptions should not be made.

One example of an expensive, irrecoverable overflow bug is the NASA spacecraft *Deep Impact*. After more than eight years in space, and multiple significant accomplishments including excavating a comet, *Deep Impact* suddenly lost contact with Earth. That was due to a 32-bit timer overflowing and causing the onboard computers to continuously reboot.

Overflow bugs can go unnoticed for many years. The binary search algorithm, which is very widely used, is often implemented incorrectly due to an overflow bug, and that bug was not noticed for two decades, in which it even made its way into the Java language's standard library.

Scheduler types

A scheduler is responsible for allocating necessary resources to do some work. To make various bits of code run on a schedule, CPU time is the resource to be allocated, and the various tasks comprise work. Different types of schedulers and different scheduling algorithms exist, with a specific choice depending on the use case and the system's constraints.

One useful concept to understand is that of *real-time systems*. Such a system has constraints, or *deadlines*, on some timings, and these deadlines are expressed in specific time measurements. A "fast response" isn't specific, "response within 2 milliseconds" is. Further, a real-time system is said to be *hard real-time* if it cannot be allowed to miss any deadlines at all, that is, a single missed deadline constitutes a system failure. Real-time systems are commonly found in embedded systems controlling aircraft, cars, industrial or medical equipment, and so on.By contrast, most consumer-oriented software isn't real-time.

A real-time system requires a scheduler that can guarantee adherence to the required deadlines. Such systems will typically run a real-time operating system (RTOS) which provides the necessary scheduling support. We're not dealing with real-time constraints, and we're not writing an operating system, so putting RTOS aside, there are two classes of schedulers to consider.

Cooperative schedulers provide *cooperative (or non-preemptive) multitasking*. In this case, every task is allowed to run until it returns control to the scheduler, at which point the scheduler can start another task. Cooperative scedulers are easy to implement, but their major downside is relying on each task to make reasonable use of CPU resources. A poorly-written task can cause the entire system to slow down or even hang entirely. Implementing individual tasks is also simpler in the cooperative case - the task can assume that it will not be interrupted, and will instead run from start to finish.

Cooperative scheduling is fairly common in low-resource embedded systems, and the implementation only requires that some kind of system-wide time exists. One example of a suitable system could be a

thermostat. It can have several tasks (measure room temperature, calculate necessary output, read user settings, log some data) running with the same periods indefinitely.

Preemptive schedulers use interrupts to *preempt*, or suspend, a task and hand control over to another task. This ensures that one task is not able to hang the entire system, or just take too long before letting other tasks run. Such schedulers implement some kind of algorithm for choosing when to interrupt a task and what task to execute next, and implementing actual preemption is another challenge.

Cooperative scheduler

To implement a basic cooperative scheduler, we don't need much code. We need to keep track of what tasks exist in the system, how often they want to run, and then the scheduler should execute those tasks. The scheduler's header file can be written so:

```
#include "systime.h"

typedef void (*task_entry_ptr)(void);

typedef struct {
    task_entry_ptr entry;
    systime_t period;
    systime_t last_run;
} task_desc;

typedef enum {
    SCHED_OK = 0,
    SCHED_TOO_MANY_TASKS
} sched_error;

#define MAX_NUM_TASKS (10u)

sched_error sched_add_task(task_entry_ptr entry, systime_t period);
void sched_run(void);
```

Each task should have an entry point, a function that returns `void` and has no parameters. A pointer to the entry point, together with the desired task period and the time of the last run, form the task descriptor in the `task_desc` type. The scheduler provides a function `sched_add_task`, which can add tasks to the scheduler at run-time. Let's look at the implementation. Here's how `sched.c` starts:

```
#include <stddef.h>
#include <stdint.h>
#include "sched.h"
```

```
4
5  static task_desc task_table[MAX_NUM_TASKS] = {0};
6  static uint_8 table_idx = 0;
7
8  sched_error sched_add_task(task_entry_ptr entry, systime_t period) {
9      if (table_idx >= MAX_NUM_TASKS) {
10         return SCHED_TOO_MANY_TASKS;
11     }
12
13     task_desc task = {
14         .entry = entry,
15         .period = period,
16         .last_run = 0
17     };
18     task_table[table_idx++] = task;
19
20     return SCHED_OK;
21 }
```

The `task_table` array is where all the task descriptors are kept. `sched_add_task` is pretty simple - if there's free space in the task table, the function creates a task descriptor and adds it to the table. The task's last run time is set to `0`. Then the interesting work happens in the scheduler's `sched_run`:

```
1  void sched_run(void) {
2      while (1) {
3          for (uint8_t i = 0; i < MAX_NUM_TASKS; i++) {
4              task_desc* task = &task_table[i];
5              if (task->entry == NULL) {
6                  continue;
7              }
8
9              if (task->last_run + task->period <= systime_get()) { /*
                   Overflow bug! */
10                 task->last_run = systime_get();
11                 task->entry();
12             }
13         }
14     }
15 }
```

NOTE

The source code from the repository for this chapter includes `sched.c` with this example, but doesn't built it by default, instead it builds `sched_preemptive.c` for the preemptive scheduler that we will cover later. Change `CMakeLists.txt` if you wish to build the cooperative scheduler.

You may have noticed that the function is contained within an infinite `while (1)` loop. The scheduler isn't supposed to terminate, and `sched_run` will be able to replace the infinite loop that we've had in `main` all along.

The actual work is done in the scheduler's `for`-loop, which loops through the entire task table, and looks for tasks whose time to run has come, that is, the system time has increased by at least `period` since the task's last run time stored in `last_run`. When the scheduler finds such a task, it updates the last run time and executes the task by calling its entry point.

All that remains in order to test the scheduler is to add a couple of tasks, and schedule them in `main`.

```
void task0(void) {
    systime_t start = systime_get();
    uart_write("Entering task 0... systime ");
    uart_write_uint(start);
    uart_write("\n");
    while (start + 1000u > systime_get());
    uart_write("Exiting task 0...\n");
}

void task1(void) {
    systime_t start = systime_get();
    uart_write("Entering task 1... systime ");
    uart_write_uint(start);
    uart_write("\n");
    while (start + 1000u > systime_get());
    uart_write("Exiting task 1...\n");
}
```

The above defines two tasks. Note how there's nothing special about those functions, it's sufficient for them to be `void` functions with no parameters for them to be scheduled with our implementation. Both tasks have the same behavior - print a message with the current system time, wait for `1000` system time ticks and exit with another message. To actually schedule them and hand control over to the scheduler, modify `main` to get rid of the infinite loop and instead do:

```
(void)sched_add_task(&task0, 5000u);
(void)sched_add_task(&task1, 2000u);

sched_run();
```

That will schedule task 0 to run every 5000 ticks (roughly 5 seconds) and task 1 for every 2000 ticks.

NOTE

The tasks use `uart_write_uint` to print the current systime, a new function not part of the previously-written UART driver. We cannot use standard C library functions such as `sprintf` without performing additional work, so this new function is a quick way to output unsigned numbers like systime. For completeness, here's the implementation:

```
void uart_write_uint(uint32_t num) {
    char buf[8];
    int8_t i = 0;
    do {
        uint8_t remainder = num % 10;
        buf[i++] = '0' + remainder;
        num /= 10;
    } while (num != 0);
    for (i--; i >= 0; i--) {
        uart_putchar(buf[i]);
    }
}
```

Running the system now should produce some output indicating the scheduler's hard at work.

```
Welcome to Chapter 8, Scheduling!
Entering task 1... systime 2000
Exiting task 1...
Entering task 1... systime 4000
Exiting task 1...
Entering task 0... systime 5000
Exiting task 0...
Entering task 1... systime 6000
Exiting task 1...
Entering task 1... systime 8000
Exiting task 1...
Entering task 0... systime 10000
Exiting task 0...
Entering task 1... systime 11000
Exiting task 1...
Entering task 1... systime 13000
Exiting task 1...
Entering task 0... systime 15000
```

```
Exiting task 0...
Entering task 1... systime 16000
Exiting task 1...
```

What does that tell us? Well, the scheduler seems to be working fine. Task 0 first gets executed at systime `5000`, which is enough time for task 1 to run twice, starting at times `2000` and `4000`. We can also see that this is very much not a real-time system, as the schedule we've provided serves as a suggestion for the scheduler but isn't strictly enforced. At systime `10000`, it's time for both tasks to be executed (task 0 for the 2nd time and task 1 for the 5th time), but task 0 gets the execution slot (due to having its entry earlier in the task table), and task 1 gets delayed until systime `11000`, when task 0 finishes.

In the scheduler's `sched_run` loop, you may have noticed a comment about an overflow bug on one line. After saying how spacecraft can get lost due to overflows, it wouldn't feel right to omit an example.

```
if (task->last_run + task->period <= systime_get()) { /* Overflow bug!
    */
```

The normal case for that line is straightforward - if at least `period` ticks have passed since `last_run`, the task needs to be run. How about when some of these variables approach `UINT32_MAX`, the maximum value they can hold? Suppose `last_run` is almost at the maximum value, e.g. `UINT32_MAX - 10` (that's `0xFF FF FF F5`). Let's say `period` is `100`. So the task ran at `UINT32_MAX - 10` ticks, took some time to complete, and the scheduler loop runs against at, for instance, systime `UINT32_MAX - 8`. Two ticks have passed since the last execution, so the task shouldn't run. But the calculation `last_run + period` is `UINT32_MAX - 10 + 100`, which overflows! Unsigned integers wrap around to zero on overflow, and so the result becomes `89`. That is less than the current system time, and the task will run again. And the problem will repeat in the next iteration as well, until eventually fixing itself after system time also overflows and wraps around to zero.

That's a potentially serious bug that is much easier to introduce than to notice. How to perform that calculation safely, then? Instead of adding the timestamps, you should use subtraction so that the intermediate result is a duration. That check should be:

```
if (systime_get() - task->last_run >= task->period)
```

It might seem like such a calculation can go wrong. Supppose `last_run` is close to overflow, as in the example before, and systime has recently rolled over. So you'd be subtracting a very large `last_run` from the small positive result of `systime_get`, which would result in a large negative number under

ordinary arithmetic. But with unsigned integers, that will still result in a correct calculation. So if the above calculation amounts to something like:

```
1 if (20 - (UINT32_MAX - 10) >= 100)
```

the left side will evaluate to `31`, and the `if` will evaluate to false.

Mathematically speaking, unsigned integers in C implement arithmetics modulo the type's maximum value plus one. So a `uint8_t`, holding the maximum value `255`, performs all operations modulo `256`. Importantly, only unsigned integers have overflow behavior that is guaranteed by the C standard. Signed integer overflow is undefined behavior, and you should avoid writing code that relies on signed integer overflows.

Life is good when you have a working scheduler, and indeed this kind of simple scheduler is pretty similar to what some embedded systems run in the real world. There are small improvements that can be made to the scheduler without fundamentally changing it, such as allowing each task to have a priority, so that the highest-priority task would be chosen at times when multiple tasks wish to run, such as at systime `10000` above. Note the low memory overhead of this scheduler. It uses 1 byte for the `table_idx` variable and creates a 12-byte task descriptor for each task. This kind of memory use is one of the reasons why such simple cooperative schedulers are a valid choice for resource-constrained embedded systems.

It is also easy to introduce some problems that a cooperative scheduler won't be able to manage well. For example, create a third task that just runs a `while (1);` loop. The scheduler will run the task and the system will hang with no hope of recovery. In a real system, a watchdog would probably be configured and reset everything, but a reset loop is not much better than a plain hang.

Summary

In this chapter, we looked at the simplest way of making a bare-metal system perform some useful work on a schedule. Doing this required a small new driver, and we built an abstract system time on top of it. From there, it was just a few more steps to have a working scheduler that runs predefined tasks on a predefined schedule.

A real-world cooperative scheduler would be slightly more complex, but not by much. For all its simplicity, a cooperative scheduler is appropriate in simple embedded systems that don't have to quickly react to external input, and have well-known tasks that don't interfere with one another and aren't expected to run for long. At the same time, the scheduler is anything but robust - a single error in a task can cause it to hang permanently.

We also saw how overflow errors are easy to introduce, and happen to often be associated with timing code.