Beginning STM32

Developing with FreeRTOS,
libopencm3 and GCC

—

Warren Gay

Apress®

Beginning STM32

Developing with FreeRTOS, libopencm3 and GCC

Warren Gay

Apress®

Beginning STM32: Developing with FreeRTOS, libopencm3 and GCC

Warren Gay
St. Catharines, Ontario, Canada

For Jackie

Table of Contents

About the Author

Warren Gay started out in electronics at an early age, dragging discarded TVs and radios home from public school. In high school he developed a fascination for programming the IBM 1130 computer, which resulted in a career-plan change to software development. Since graduating from Ryerson Polytechnical Institute, he has enjoyed a 30-plus-year software developer career, programming mainly in C/C++. Warren has been programming Linux since 1994 as an open source contributor and professionally on various Unix platforms since 1987.

Before attending Ryerson, Warren built an Intel 8008 system from scratch before there were CP/M systems and before computers got personal. In later years, Warren earned an advanced amateur radio license (call sign VE3WWG) and worked the amateur radio satellites. A high point of his ham-radio hobby was making digital contact with the Mir space station (U2MIR) in 1991.

Warren works at Datablocks.net, an enterprise-class ad-serving software services company where he programs C++ server solutions on Linux back-end systems.

About the Technical Reviewer

Stewart Watkiss is a keen maker with a particular interest in physical computing. He earned a master's degree in electronic engineering from the University of Hull in 1996 and a master's degree in computer science from the Georgia Institute of Technology in 2017.

Most of his projects are based around the Raspberry Pi, which he creates alone or together with his two children. He has also created projects based around the Arduino and other platforms. Many of his projects are available on his website, www.penguintutor.com, which also provides tutorials on Linux and electronics. He is the author of the book *Learn Electronics with Raspberry Pi*, published by Apress.

Stewart also volunteers as a STEM Ambassador, working with local schools and educational events to enthuse children about programming and physical computing.

Introduction

There is considerable interest in the ARM Cortex platform today because ARM devices are found everywhere. Units containing ARM devices range from the small microcontroller embedded systems to cellphones and larger servers running Linux. Soon, ARM will also be present in higher numbers in the datacenter. These are all good reasons to become familiar with ARM technology.

With the technology ranging from microcontrollers to full servers, the question naturally arises: "Why study embedded device programming? Why not focus on end-user systems running Linux, like the Raspberry Pi?"

The simple answer is that embedded systems perform well in scenarios that are awkward for larger systems. They are frequently used to interface with the physical world. They go between the physical world and a desktop system, for example. The humble keyboard uses a dedicated MCU (microcontroller unit) to scan key switches of the keyboard and report key-press events to the desktop system. This not only reduces the amount of wiring necessary but also frees the main CPU from expending its high-performance computing on the simple task of noticing key-press events.

Other applications include embedded systems throughout a factory floor to monitor temperature, security, and fire detection. It makes little sense to use a complete desktop system for this type of purpose. Stand-alone embedded systems save money and boot instantly. Finally, the MCU's small size makes it the only choice in flying drones where weight is a critical factor.

The development of embedded systems traditionally required the resources of two disciplines:

- Hardware engineer
- Software developer

Frequently, one person is assigned the task of designing the end product. Hardware engineers specialize in the design of the electronic circuits involved, but eventually the

1

product requires software. This can be a challenge because software people generally lack the electronics know-how while the engineers often lack the software expertise. Because of reduced budgets and delivery times, the electronics engineer often becomes the software engineer as well.

There is no disadvantage to one person's performing both design aspects as long as the necessary skills are present. Whether you're an electronics engineer, software developer, hobbyist, or maker, there is nothing like real, down-to-earth practice to get you going. That is what this book is all about.

STM32F103C8T6

The device chosen for this book is the STMicroelectronics STM32F103C8T6. This part number is a mouthful, so let's break it down:

- STM32 (STMicroelectronics platform)

- F1 (device family)

- 03 (subdivision of the device family)

- C8T6 (physical manifestation affecting amount of SRAM, flash memory, and so on)

As the platform name implies, these devices are based upon a 32-bit path and are considerably more powerful than 8-bit devices as a result.

The F103 is one branch (F1 + 03) of the STM32 platform. This subdivision decides the CPU and peripheral capabilities of the device.

Finally, the C8T6 suffix further defines the capabilities of the device, like the memory capacity and clock speeds.

The STM32F103C8T6 device was chosen for this book because of the following factors:

- *very* low cost (as low as $2 US on eBay)

- availability (eBay, Amazon, AliExpress, etc.)

- advanced capability

- form factor

The STM32F103C8T6 is likely to remain the lowest-cost way for students and hobbyists alike to explore the ARM Cortex-M3 platform for quite some time. The device is readily available and is extremely capable. Finally, the form factor of the small PCB allows header strips to be soldered to the edges and plugged into a breadboard. Breadboards are the most convenient way to perform a wide array of experiments.

The MCU on a blue PCB (Figure 1-1) is affectionately known as the "Blue Pill," inspired by the movie *The Matrix*. There are some older PCBs that were red in color and were referred to as the "Red Pill." There are still others, which are black and are known as the "Black Pill." In this book, I'll be assuming you have the Blue Pill model. Apart from some USB deficiencies, there should be little other difference between it and the other models.

Figure 1-1. *The STM32F103C8T6 PCB (printed circuit board) with the header strips soldered in, often referred to as the "blue pill"*

Low cost has another advantage—it allows you to own *several* devices for projects involving CAN communications, for example. This book explores CAN communication using three devices connected by a common bus. Low cost means not being left out on a student budget.

The peripheral support of the STM32F103 is simply amazing when you consider its price. Peripherals included consist of:

- 4 x 16-bit GPIO Ports (most are 5-volt tolerant)

- 3 x USART (Universal Synchronous/Asynchronous Receiver/ Transmitter)

- 2 x I2C controllers

- 2 x SPI controllers

- 2 x ADC (Analog Digital Converter)

- 2 x DMA (Direct Memory Address controllers)

- 4 x timers

- watch dog timers

- 1 x USB controller

- 1 x CAN controller

- 1 x CRC generator

- 20K static RAM

- 64K (or 128K) FLASH memory

- ARM Cortex M3 CPU, max 72 MHz clock

There are some restrictions, however. For example, the USB and CAN controllers cannot operate at the same time. Other peripherals may conflict over the I/O pins used. Most pin conflicts are managed through the AFIO (Alternate Function Input Output) configuration, allowing different pins to be used for a peripheral's function.

In the peripheral configuration, several separate clocks can be individually enabled to tailor power usage. The advanced capability of this MCU makes it suitable for study. What you learn about the STM32F103 family can be leveraged later in more advanced offerings like the STM32F407.

The flash memory is officially listed at 64K bytes, but you may find that it supports 128K. This is covered in Chapter 2 and permits good-sized applications to be flashed to the device.

FreeRTOS

Unlike the popular AVR family of chips (now owned by Microchip), the STM32F103 family has enough SRAM (static RAM) to comfortably run FreeRTOS (`freertos.org`). Having access to a RTOS (real-time operating system) provides several advantages, including

- preemptive multitasking;
- queues;
- mutexes and semaphores; and
- software timers.

Of particular advantage is the multitasking capability. This eases the burden of software design considerably. Many advanced Arduino projects are burdened by the use of state machines with an *event loop model*. Each time through the loop, the software must poll whether an event has occurred and determine if it is time for some action. This requires management of state variables, which quickly becomes complex and leads to programming errors. Conversely, preemptive multitasking provides separate control tasks that clearly implement their independent functions. This is a proven form of software abstraction.

FreeRTOS provides preemptive multitasking, which automatically shares the CPU time among configured tasks. Independent tasks, however, do add some responsibility for safely interacting between them. This is why FreeRTOS also provides message queues, semaphores, mutexes, and more to manage that safely. We'll explore RTOS capabilities throughout this book.

libopencm3

Developing code for MCU applications can be demanding. One portion of this challenge is developing with the "bare metal" of the platform. This includes all of the specialized peripheral registers and their addresses. Additionally, many peripherals require a certain "dance" to make them ready for use.

This is where libopencm3 fits in (from `libopencm3.org`). Not only does it define the memory addresses for the peripheral register addresses, but it also defines macros for special constants that are needed. Finally, the library includes tested C functions for interacting with the hardware peripheral resources. Using libopencm3 spares us from having to do all of this from scratch.

No Arduino

There is no Arduino code presented in this book. Arduino serves its purpose well, allowing students to wade into the MCU world without prior knowledge. This book, however, is targeted to go beyond the Arduino environment using a professional mode of development independent of Arduino tools.

Without Arduino, there is no "digital port 10." Instead, you work directly with an MCU port and optionally a pin. For example, the Blue Pill device used in this book has the built-in LED on port C, as pin 13. Operating directly with ports permits I/O operations with all 16 pins at one time when the application needs it.

No IDE

There was a conscious decision to choose for this book a development environment that was neutral to your desktop development platform of choice. There are a number of Windows-based IDE environments available, with varying licenses. But IDEs change, licenses change, and their associated libraries change with time. The advantage of the given IDE is often discarded when the IDE and the operating system it runs upon change.

Using a purely open sourced approach has the advantage that you are shielded from all this version churn and burn. You can mothball all of your code and your support tools, knowing that they can all be restored to operation ten years from now, if required. Restoring licensed software, on the other hand, leaves you vulnerable to expired licenses or online sites that have gone dark.

This book develops projects based upon the following open sourced tools and libraries:

- gcc/g++ (GNU compiler collection: open sourced)

- make (GNU binutils: open sourced)

- libopencm3 (library: open sourced)

- FreeRTOS (library: open source and free for commercial use)

With this foundation, the projects in this book should remain usable long after you purchase this book. Further, it permits Linux, FreeBSD, and MacOS users—in addition to those using the Windows platform—to use this book. If you do use Windows, you may

want to download and install the Cygwin environment (`www.cygwin.com`) because a Linux-like environment is assumed for the demo project builds.

All of the projects presented make use of the GNU (GNU is not Unix) `make` utility, which provides several build functions with minimum effort. If the provided builds in this book present errors, then make sure to use the GNU `make` command, especially on FreeBSD. Some systems install GNU `make` as `gmake`.

Development Framework

While it is possible to make gcc, libopencm3, and FreeRTOS work together on your own, it does require a fair amount of organization and effort. How much is your time worth?

Rather than do this tedious work, a development framework is available for free from `github.com` for download. This framework integrates libopencm3 with FreeRTOS for you. Also provided are the make files needed to build the whole project tree at once or each project individually. Finally, there are some open source library routines included that can shorten the development time of your new applications. This framework is included as a `github.com` download or with the book's own source code download.

Assumptions About You

This book is targeted to an audience wanting to go beyond the Arduino experience. This applies to hobbyists, makers, and engineers alike. The software developed in this book uses the C programming language, so fluency there will be helpful. Likewise, some basic digital electronics knowledge is assumed as it pertains to the peripheral interfaces provided by the platform. Additional light theory may be found in areas like the CAN bus, for example.

The STM32 platform can be a challenge to configure and to get operating correctly. Much of this challenge is the result of the extreme *configurability* of the peripheral devices. Each portion depends upon a clock, which must be enabled and divisor configured. Some devices are further affected by upstream clock configurations. Finally, each peripheral itself must be enabled and configured for use. You won't have to be an expert, because these ducks-in-a-row procedures will be laid out and explained in the chapters ahead.

Hobbyists and makers need not find the book difficult. Even when challenged, they should be able to build and *run* each of the project experiments. As knowledge and confidence builds, each reader can grow into the topics covered. As part of this exploration, all readers are encouraged to modify the projects presented and run further experiments. The framework provided will also allow you to create new ready-to-go projects with a minimum of effort.

What You Need

Let's briefly cover some items that you might want to acquire. Certainly, number one on the list is the Blue Pill device (see Figure 1-1). I recommend that you purchase units that include the header strips to be soldered onto the PCB so that you can easily use the unit on a breadboard (or presoldered, if you prefer).

These units are Buy-it-Now priced on eBay at around $2.13 US, with free shipping from various sellers. To find these deals, simply use the part number STM32F103C8T6 for your search. Chapters 18–19 use three of these units communicating with each other over a CAN bus. If you'd like to perform those experiments, be sure to obtain at least three units. Otherwise, the demo projects only involve one unit at a time. A spare is always recommended in case of an unhappy accident.

ST-Link V2 Programming Unit

The next essential piece of hardware is a programming adapter. Fortunately, these are also very economically priced. These can be found on eBay for about $2.17 US, with free shipping. Simply search for "ST-Link." Be sure to get the "V2" programmer since there is no point in using the inferior older unit.

Most auctions will include four detachable wires to connect the unit to your STM32 device. Try to buy a unit that includes these unless you already have a cable. Figure 1-2 illustrates the USB programmer, usable from Windows, Raspberry Pi, Linux, MacOS, and FreeBSD.
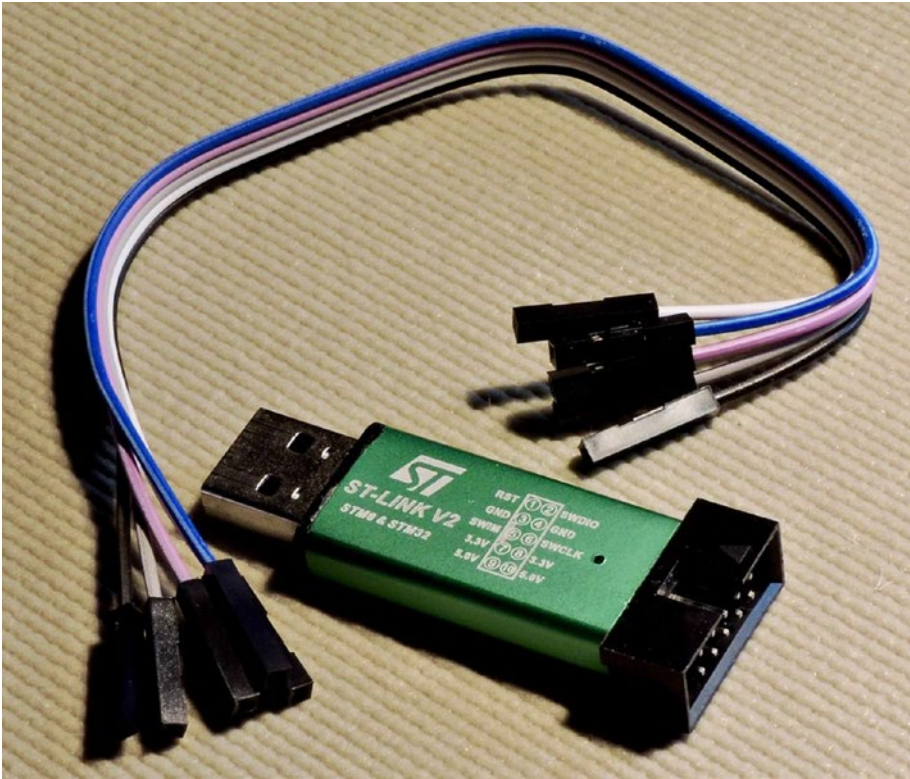
Figure 1-2. *ST-Link V2 programmer and cable*

The STM32F103C8T6 device can be programmed in multiple ways, but this book will only use the ST-Link V2 USB programmer. This will simplify things for you when doing project development and allows remote debugging.

A USB extension cable is useful with this unit. If you don't have one, you might consider getting one.

Breadboard

This almost goes without saying, but a breadboard is necessary to prototype experiments. The breadboard is a solderless way to quickly wire up experiments, try them, and then pull out the wires for the next experiment.

Many of the projects in this book are small, requiring space for one Blue Pill device and perhaps some LEDs or a chip or two. However, other experiments, like the one in Chapters 18–19, use three units communicating with each other over a CAN bus.

9

I recommend that you obtain a breadboard that will fit four units (this leaves a little extra hookup space). Alternatively, you could simply buy four small breadboards, though this is less convenient.

Figure 1-3 illustrates the breadboard that I am using in this book. It is not only large enough, but also has power rails at the top and bottom of each strip. The power rails are recommended, since this eases the wiring.
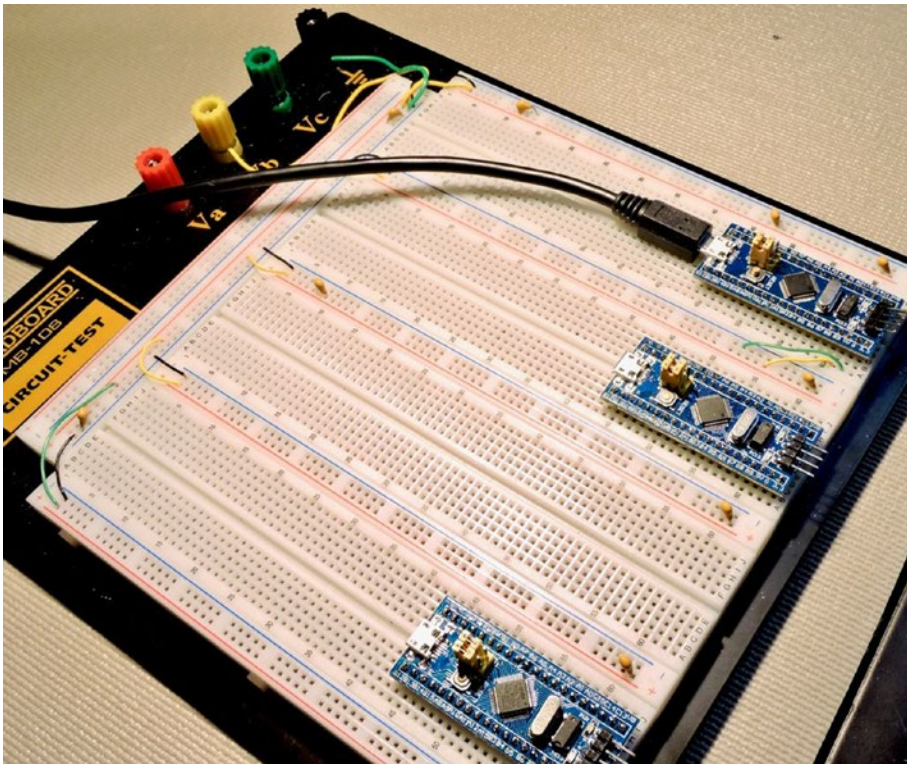
Figure 1-3. *A breadboard with supply rails*

DuPont (Jumper) Wires

You might not give much thought to the wiring of a breadboard, but you will find that DuPont wires can make a huge difference. Yes, you can cut and strip your own AWG22 (or AWG24) gauge wires, but this is inconvenient and time consuming. It is far more convenient to have a small box of wires ready to go. Figure 1-4 illustrates a small random collection of DuPont wires.
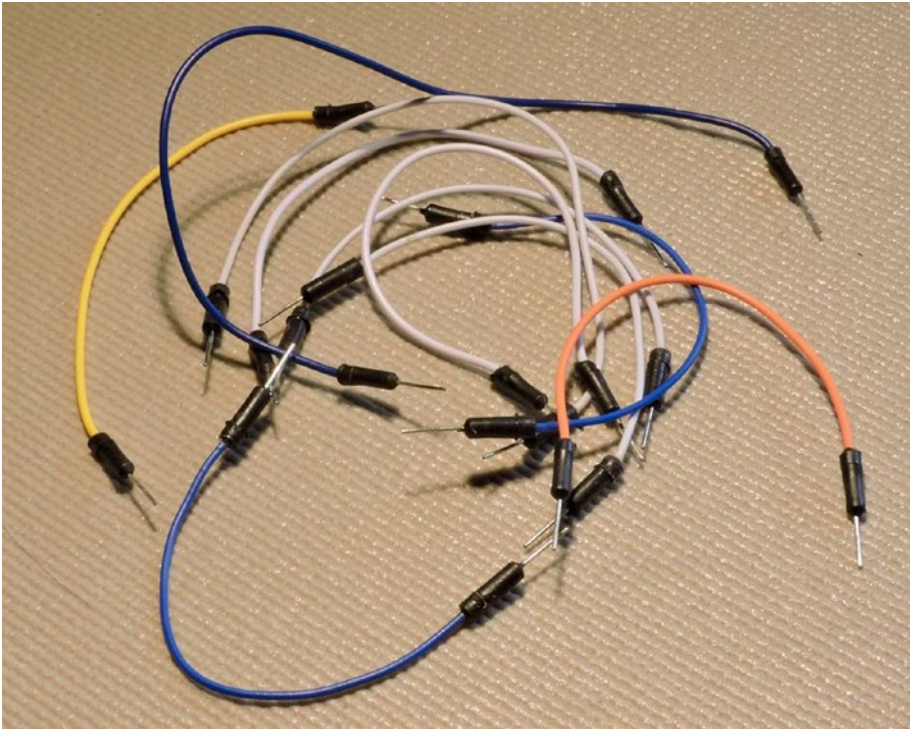
Figure 1-4. *A random collection of DuPont wires*

Male-to-male DuPont wires can be purchased in assorted sets on eBay for about the Buy-it-Now price of $2.00 US with free shipping. They might have auction titles like "65Pcs Male to Male Solderless Flexible Breadboard DuPont Jumper Cable Wires." I recommend that you get the *assorted* sets so that you get different colors and lengths. A search like "DuPont wires male -female" should yield good results. The "-female" keyword will eliminate any ads that feature female connectors.

0.1 uF Bypass Capacitors

You might find that you can get by without bypass caps (capacitors), but they are recommended (shown in Figure 1-5 as yellow blobs on the power rails). These can be purchased in quantity from various sources, including eBay.
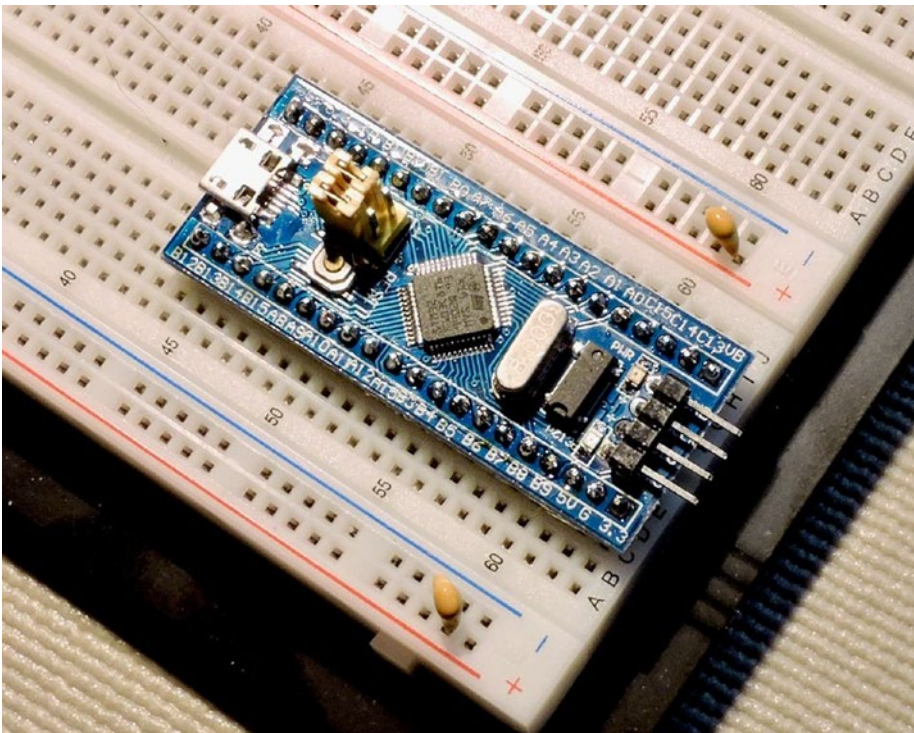
Figure 1-5. *Breadboard with STM32F103C8T6 and 0.1 uF bypass capacitors installed on the rails*

Try to buy quality capacitors like Metalized Polyester Film units if possible. The voltage rating can be as low as 16 volts. A few of these should be plugged into your supply rails on the breadboard, between the positive and negative rails, to filter out any voltage transients and noise.

USB TTL Serial Adapter

This device is essential for some of the projects in this book. Figure 1-6 illustrates the unit that I used. This serial adapter is used to communicate data to your desktop/laptop. Without a display, this allows you to communicate through a virtual serial link (via USB) to a terminal program.

There are several types of these available on eBay and elsewhere, but be careful to get a unit with *hardware flow control signals*. The cheapest units will lack these additional signals (look for RTS and CTS). Without hardware flow control signals, you will not be able to communicate at high speeds, such as 115200 baud, without losing data.

If you're running Windows, also be careful of buying FTDI (FTDI Chip) fakes. There were reports of FTDI software drivers bricking the fake devices at one time. Your choice doesn't have to include FTDI, but if the device claims FTDI compatibility, be aware and check your driver support.

You'll notice in Figure 1-6 that I have a tag tied to the end of the cable. That tag reminds me which colored wire is which so that I can hook it up correctly. You might want to do something similar.
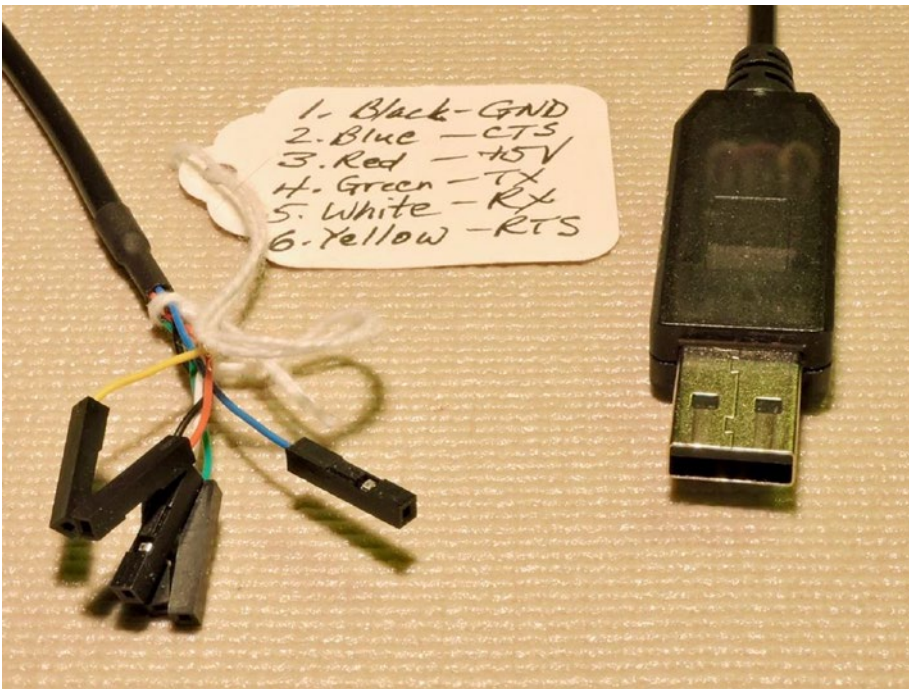
Figure 1-6. *A USB-to-TTL serial (5V) adapter cable*

These are normally 5-volt devices and are hence TTL compatible. Note, however, that one of the features of the STM32F103 family of devices is that many of the GPIO pins are 5-volt tolerant, even though the MCU operates from a +3.3-volt supply. This permits the use of these TTL adapters without causing harm. More will be said about this later. Other units can be purchased that operate at the 3.3-volt level or that can switch between 5 and 3.3 volts.

Power Supply

Most of the projects presented will run just fine off of the USB or TTL adapter power output. But if your project draws more than the usual amount of current, then you may need a power adapter. Figure 1-7 illustrates a good adapter to fit the breadboard power rails. It can be purchased from eBay for about $1.00 US with free shipping. Mine was advertised as "MB102 Solderless Breadboard Power Supply Module, 3.3V 5V for Arduino PCB Board." If your breadboard lacks power rails, you may need to shop for a different type of breadboard.
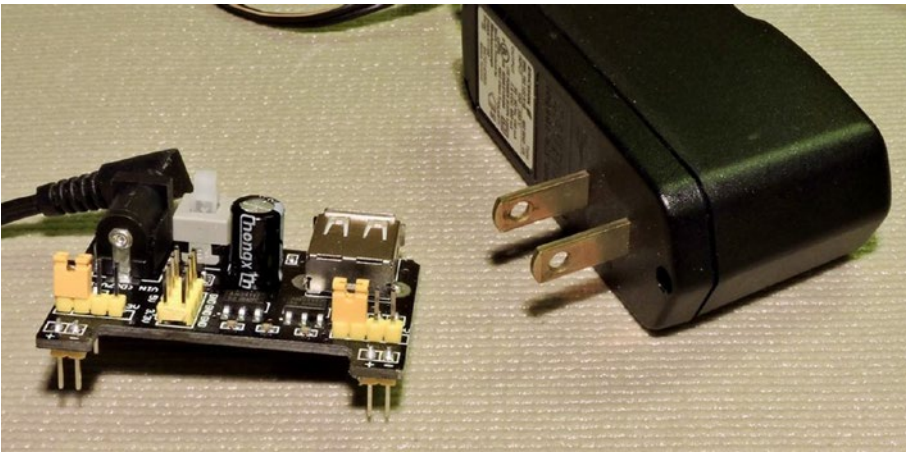
Figure 1-7. *A small breadboard power supply and 7.5 VDC adapter*

The MB102 is convenient because it can be jumpered to supply 3.3 or 5 volts. Additionally, it includes a power on/off button.

The other consideration is the *wall adapter* to supply the *input* power (this is not included). While the MB102 accepts up to 12 volts of input, I found that most 9 VDC wall adapters had an open circuit voltage near 13 volts or more. I feel that those are risky because if the cheap MB102 fails for any reason, the over-voltage might leak through and damage your MCU unit(s) as well.

Foraging through my junk box of "wall warts," I eventually found an old Ericsson phone charger rated at 7.5 VDC at 600 mA. It measured an unloaded voltage of 7.940 volts. This is much closer to the 5 and 3.3 volt outputs that the MB102 will regulate to. If you have to purchase a power adapter, I recommend a similar unit.

Small Stuff

There are some small items that you may already have. Otherwise, you will need to get some LEDs and resistors for project use. Figure 1-8 shows a random set of LEDs and a SIP-9 resistor.
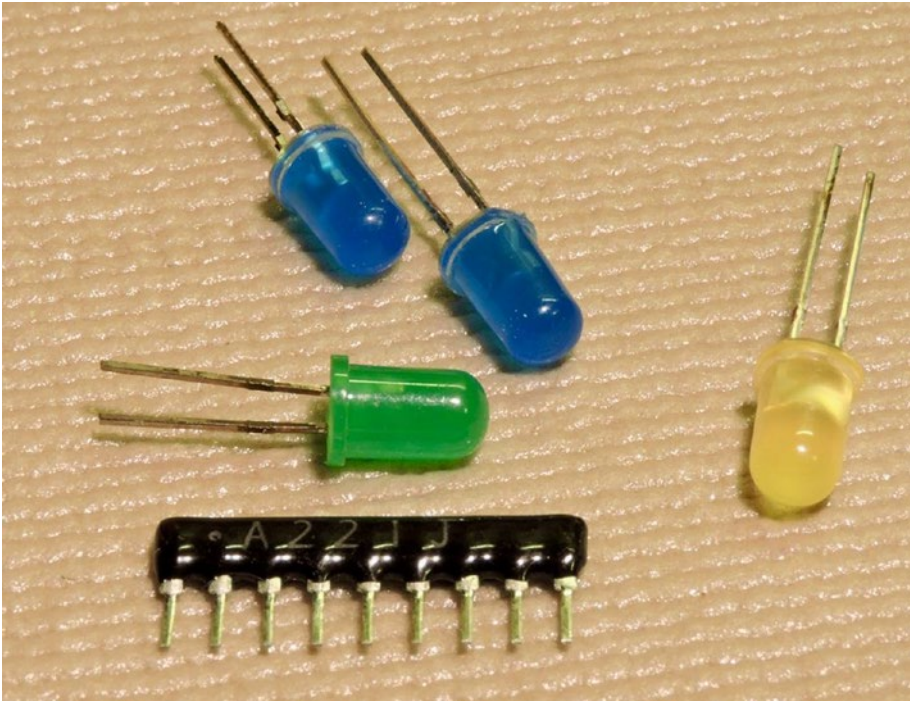
Figure 1-8. *A random collection of 5 mm LEDs and one SIP-9 resistor at the bottom*

Usually an LED is targeted for about 10 mA of current for normal brightness. Smaller LEDs only require maybe 2 to 5 mA. With a supply voltage near 3.3 volts, you'll want a resistor of about 220 Ω to limit the current (220 ohms limits the current to a maximum of approximately 7 mA). So, get a few 220 Ω resistors (1/8th watt will be sufficient).

Another part you may want to consider stocking is the SIP-9 resistor. Figure 1-9 illustrates the internal schematic for this part. If, for example, you want to drive eight LEDs, you would need eight current-limiting resistors. Individual resistors work but require extra wiring and take up breadboard space. The SIP-9 resistor, on the other hand, has one connection common to the eight resistors. The other eight connections are the other end of the internal resistors. Using this type of package, you can reduce the parts count and wiring required.
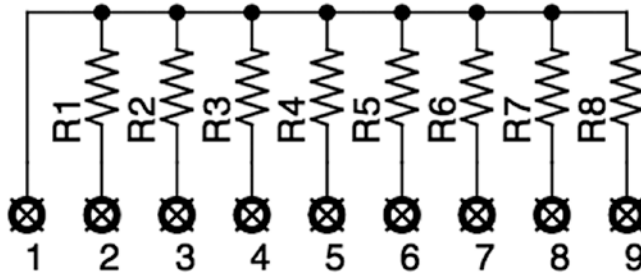
Figure 1-9. *The internal schematic view of a SIP-9 resistor*

Summary

This chapter has introduced the main actors that will appear in this book. It also itemized most of the things that you might need to acquire. The next chapter will guide you through the necessary steps of software installation. Once that is out of the way, the real fun can begin.

Software Setup

Before you can get underway with project-related work, you need some software installed. There are a number of "moving parts" involved. Despite this, the process should proceed smoothly. Once accomplished, it need not be revisited.

Directory Conventions Used

Throughout this book, I'll need to refer to different subdirectories of the supplied software. It is assumed that the top level of your installed software is named "~/stm32f103c8t6." So, when I refer to a pathname "~/stm32f103c8t6/libopencm3/README.md," I'll assume that starts from your home (~) directory (wherever that is). I'll often use this pathname convention for clarity, even though your current directory may be the correct one for the file being referenced.

Operating Software

I'm also going to assume you have a POSIX (Linux/Unix) environment from which to run commands. The Linux or Raspberry Pi environments using the bash shell are perhaps the most natural. Other good environments include FreeBSD and MacOS. From FreeBSD, I'll assume that you are using the bash shell.

If you're using Windows and you haven't installed it yet, you'll want to install Cygwin (https://www.cygwin.com). Some might use MSYS instead. After installing the base Cygwin system, make sure that you also install make and git. This will give you a Linux-like command-line environment from which to build software.

Mac users will need to install git at a minimum. You'll also need GNU make, especially if you use FreeBSD (Free Berkeley Software Distribution). Sometimes GNU make is installed as gmake instead on a BSD (Berkely Software Distribution) system. If you're using Mac Homebrew (https://brew.sh), you can install these as follows:

```
$ brew install make
$ brew install git
```

If you're a Mac Ports (https://www.macports.org) user, you'll need to use that framework to install make and git.

Book Software

The directory structure for building with libopencm3 and FreeRTOS is available from github.com. Choose a suitable place from which to create a subdirectory. This book will assume home directory:

```
$ cd ~
```

Use the git command to download and create a subdirectory as follows:

```
$ git clone https://github.com/ve3wwg/stm32f103c8t6.git
```

The preceding command will create directory ~/stm32f103c8t6. However, feel free to rename it to something easier to type, like ~/stm32.

libopencm3

Next, we must download the libopencm3 software into the correct place. First, change to the subdirectory, and then issue the git clone command for libopencm3:

```
$ cd ~/stm32f103f8t6
$ git clone https://github.com/libopencm3/libopencm3.git
```

This will populate the directory ~/stm32f103c8t6/libopencm3 with files and subdirectories.

FreeRTOS

The next important piece of software is FreeRTOS. Unfortunately, it must be downloaded and unpacked as a zip file.

1. Go to http://www.freertos.org.

2. Locate "Download Source" at the left.

3. Click the link "2. Click to download the latest official release from SourceForge."

Depending on your browser and operating system, a zip file should be downloaded automatically. It will have a version number in the file name. At the time of writing, the downloaded file name is `FreeRTOSv10.0.1.zip`. Change to the `~/stm32f103c8t6/rtos` subdirectory before unpacking the zip file. On my Mac, the download directory is `~/Downloads`. Substitute in the `unzip` command as required for your system:

```
$ cd ~/stm32f103c8t6/rtos
$ unzip ~/Downloads/FreeRTOSv10.0.1.zip
```

Once that completes, there should be several files and subdirectories under `~/stm32f103c8t62/rtos/FreeRTOSv10.0.1`.

~/stm32f103c8t6/rtos/Project.mk

Because the version number of FreeRTOS is included in the subdirectory name, there is a potential change left. Edit the file `Project.mk` with your favorite editor (or nano) and locate the following line near the top of the file:

```
FREERTOS        ?= FreeRTOSv10.0.1
```

If your version of FreeRTOS is newer than this, like `FreeRTOSv11.0.0`, then edit it to match your version and resave the file:

```
FREERTOS        ?= FreeRTOSv11.0.0
```

This will allow the `Project.mk` make file to work correctly later when you want to create a new RTOS project.

ARM Cross Compiler

If you don't yet have an ARM cross compiler installed, it will need to be installed. If you're running Linux or Raspberry Pi, you may be able to just use the `apt-get` command to install it. Despite that, I recommend that you download and install the toolchain as outlined next instead because some cross-compiler tools are not well organized and are sometimes incomplete.

If you're running Mac or Windows (Cygwin), then definitely use the following procedure. This procedure is also recommended for Linux and Raspberry Pi if you have had problems with the installed packages:

1. Go to the site https://developer.arm.com.

2. Click on the link "Linux/Open Source."

3. Scroll down and click on "ARM GNU Embedded Toolchain."

4. Scroll down and click on the big button labeled "Downloads."

5. Scroll down until you find the platform download required. Windows 32-bit, Linux 64-bit, Mac OS X 64-bit, etc. Click on the appropriate choice for your platform to download.

6. Create a system directory `/opt` (if you do not already have one):

   ```
   $ sudo -i
   # mkdir /opt
   ```

7. Change to the `/opt` directory (as root):

   ```
   # cd /opt
   ```

8. From this point, you'll unpack your compiler download (Mac example). Be sure to be specific about your home directory:

   ```
   # tar xjf ~myuserid/Downloads/gcc-arm-none-eabi-6-2017-q2-
   update-mac.tar.bz2
   ```

   Use tar option "`j`" if the ending of the file is `.bz2`. Otherwise, use "`z`" when the ending is `.gz`. If you don't have the GNU `tar` command installed on the Mac, then you can install it using macports (www.macports.org) or Homebrew (https://brew.sh/).

9. Once the tar file has been extracted, it may produce a large directory name like `gcc-arm-none-eabi-6-2017-q2-update`. Now is a good time to shorten that:

   ```
   # mv gcc-arm-none-eabi-6-2017-q2-update gcc-arm
   ```

   This will rename the directory /opt/gcc-arm-none-eabi-6-2017-q2-update to a more manageable name /opt/gcc-arm.

10. Now, exit root and return to your developer session. In that session, add the compiler's bin directory to your PATH:

    ```
    $ export PATH="/opt/gcc-arm/bin:$PATH"
    ```

11. At this point, you should be able to test your cross compiler:

    ```
    $ arm-none-eabi-gcc --version
    arm-none-eabi-gcc (GNU Tools for ARM Embedded Processors
    6-2017-q2-update) 6.3.1 20170620 (release) [ARM/embedded-
    6-branch revision 249437]
    Copyright (C) 2016 Free Software Foundation, Inc.
    This is free software; see the source for copying
    conditions. There is NO
    warranty; not even for MERCHANTABILITY or FITNESS FOR A
    PARTICULAR PURPOSE.
    ```

If the compiler doesn't start and instead gives you a message like this:

```
$ arm-none-eabi-gcc --version
-bash: arm-none-eabi-gcc: command not found
```

then your PATH variable is either not set up properly or not exported, or the installed tools are using a different prefix. Perform the following if necessary (the output has been abbreviated slightly here):

```
$ ls -l /opt/gcc-arm/bin
total 75128
-rwxr-xr-x@ 1 root  wheel  1016776 21 Jun 16:11 arm-none-eabi-addr2line
-rwxr-xr-x@ 2 root  wheel  1055248 21 Jun 16:11 arm-none-eabi-ar
-rwxr-xr-x@ 2 root  wheel  1749280 21 Jun 16:11 arm-none-eabi-as
```

```
-rwxr-xr-x@ 2 root  wheel  1206868 21 Jun 19:08 arm-none-eabi-c++
-rwxr-xr-x@ 1 root  wheel  1016324 21 Jun 16:11 arm-none-eabi-c++filt
-rwxr-xr-x@ 1 root  wheel  1206788 21 Jun 19:08 arm-none-eabi-cpp
-rwxr-xr-x@ 1 root  wheel    42648 21 Jun 16:11 arm-none-eabi-elfedit
-rwxr-xr-x@ 2 root  wheel  1206868 21 Jun 19:08 arm-none-eabi-g++
-rwxr-xr-x@ 2 root  wheel  1202596 21 Jun 19:08 arm-none-eabi-gcc
...
-rwxr-xr-x@ 2 root  wheel  1035160 21 Jun 16:11 arm-none-eabi-nm
-rwxr-xr-x@ 2 root  wheel  1241716 21 Jun 16:11 arm-none-eabi-objcopy
...
```

If you obtained your cross compiler from a different source than the one indicated, you might not have the prefix names. If you see the file name gcc instead of arm-none-eabi-gcc, you'll need to invoke it as simply gcc. *But be careful in this case*, because your cross compiler may get confused with your *platform* compiler. The prefix arm-none-eabi- prevents this. When you go to use your cross platform gcc, check that the correct compiler is being used with the type command:

```
$ type gcc
arm-none-eabi-gcc is hashed (/opt/gcc-arm/bin/gcc)
```

If your bash shell is locating gcc from a different directory than the one you installed, then your PATH is not set correctly.

If you must change the toolchain prefix, then the top-level ~/stm32f103c8t6/ Makefile.incl should be edited:

```
$ cd ~/stm32f103c8t6
$ nano Makefile.incl
```

Modify the following line to suit and resave it:

```
PREFIX          ?= arm-none-eabi
```

In a normal situation where the cross-platform prefix is used, you should also be able to make this confirmation:

```
$ type arm-none-eabi-gcc
arm-none-eabi-gcc is hashed (/opt/gcc-arm/bin/arm-none-eabi-gcc)
```

This confirms that the compiler is being run from the installed `/opt/gcc-arm` directory.

Note   The `PATH` variable will need modification for each new terminal session to use the cross-compiler toolchain. For convenience, you may want to create a script, modify your `~/.bashrc` file, or create a shell alias command to do this.

Build the Software

At this point, you've installed the book software, libopencm3, FreeRTOS, and the ARM cross-compiler toolchain. With the `PATH` variable set (as just seen), you should now be able to change to your `stm32f103c8t6` directory and type `make` (some users might need to use `gmake` instead):

```
$ cd ~/stm32f103c8t6
$ make
```

This will build `~/stm32f103c8t6/libopencm3` first, followed by all other subdirectories.

There is always the possibility that a new version of libopencm3 might create build problems. These are difficult to anticipate, but here are some possibilities and solutions:

1. Something in libopencm3 is flagged as an error by the cross compiler, where previously it was acceptable. You can:

   a. Correct or work around the problem in the libopencm3 sources.

   b. Try a later (or prior) version of the cross-compiler toolchain. Newer toolchains will often correct the issue. For reference, the toolchain used for this book was "GNU Tools for ARM Embedded Processors 6-2017-q2-update) 6.3.1 20170620."

   c. Install an older version of libopencm3. All projects tested in this book used the library with the latest git commit dated October 12, 2017.

2. Something in the book's software is busted. Check the git repository for updates. As issues become known, fixes will be applied and released there. Also check the top-level `README.md` file.

ST-Link Tool

There is one final piece of software that may need installation. If you've not already installed it using your system's package manager, you'll need to install it now. Even if you have it installed already, it may be outdated. Let's test it to see:

```
$ st-flash
```

Look for the following line in the help display:

```
./st-flash [--debug] [--reset] [--serial <serial>] [--format <format>] \
  [--flash=<fsize>] {read|write} <path> <addr> <size>
```

If you don't see the option `--flash=<fsize>` mentioned, then you may want to download the latest from github and build it from source code. This is only necessary if you want to use more than 64K of flash memory. None of the demos in this book go over that limit.

People have reported that many of the STM32F103C8T6 units support 128K of flash memory, even though the device reports that it only has 64K. The following command probes a unit that I own, from eBay, for example:

```
$ st-info --probe
Found 1 stlink programmers
 serial: 493f6f06483f53564554133f
openocd: "\x49\x3f\x6f\x06\x48\x3f\x53\x56\x45\x54\x13\x3f"
  flash: 65536 (pagesize: 1024)
   sram: 20480
 chipid: 0x0410
  descr: F1 Medium-density device
```

The information reported indicates that the device only supports 65536 bytes (64K) of flash. Yet, I know that I can flash up to 128K and use it (all of mine support 128K). It has been suggested that both the F103C8 devices and the F103B8 devices use the same silicon die. I'll cover using the ST-Link V2 programmer on your device in the next chapter.

If you don't have these utilities installed, do so now using apt-get, brew, yum, or whatever your package manager is. Failing a package install, you can download the latest source code from github here:

```
$ cd ~
$ git clone https://github.com/texane/stlink.git
$ cd ./stlink
$ make
$ cd build/Release
$ sudo make install
```

If you run into trouble with this, see the following online resources:

- The README.md file at https://github.com/texane/stlink

- https://github.com/texane/stlink/blob/master/doc/compiling.md

- Make sure that you have libusb installed.

- Some Linux distributions may require you to also perform sudo ldconfig after the install.

Summary

With the software installs out of the way, we can finally approach the hardware and do something with it. In the next chapter, we'll look at your power options and then apply the ST-Link V2 programmer to probe your device.

Power Up and Blink

The unit that you purchased has likely already been preprogrammed to blink when it is powered up (perhaps you've checked this already). This makes it easy to test that it is a working unit. There are a few other important details regarding power, reset, and LEDs that need to be discussed in this chapter. Finally, the use of the ST-Link V2 programmer and a device probe will be covered.

Power

The STM32F103C8T6 PCB, otherwise known as the "Blue Pill" board, has a number of connections, including a few for power. It is not necessary to use all of the power connections at once. In fact, it is best to use only one set of connections. To clarify this point, let's begin with an examination of your power options. Figure 3-1 illustrates the connections around the edges of the PCB, including power.
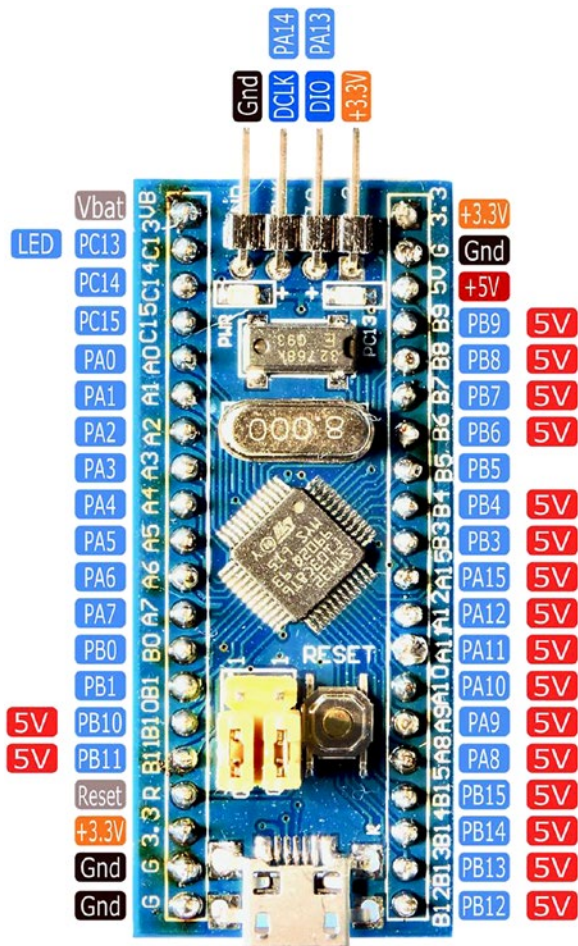
Figure 3-1. *Power and GPIO connections to the STM32F103C8T6 "Blue Pill" PCB. Power can be supplied to a +5V, +3.3V, or USB port, with the matching voltage. Pins marked as "5V" (with no plus sign) are 5-volt tolerant inputs. Pins marked with a plus sign are for power input.*

The four pins at the top end of the board (darker blue) are used for programming the device. Notice that the programming connection labeled DIO is also capable of being a GPIO PA13. Likewise, DCLK is capable of being a GPIO PA14. You'll discover how configurable the STM32 can be as we go through this book.

At the programming connector, note that the input supply voltage is +3.3 volts. This connection is electrically the same as any of the others that are labeled "+3.3V" around the PCB. These are shown in a light orange.

+3.3V Regulator

The STM32F103C8T6 chip is designed to operate from any voltage from 2 to 3.3 volts. The Blue Pill PCB provides a tiny +3.3-volt regulator labeled "U1" on the underside (see Figure 3-2). My unit used a regulator with an SMD code of 4A2D, which is an XC6204 series part. Yours may vary.
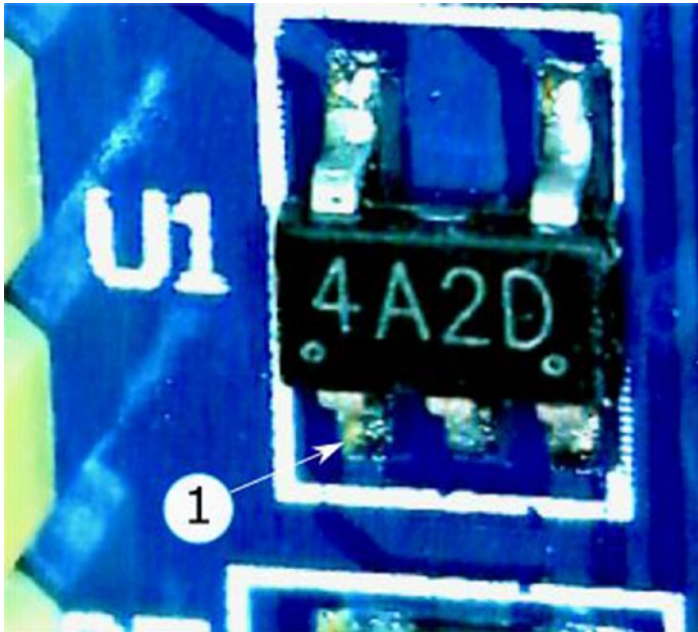
Figure 3-2. *The +3.3-volt regulator on the underside of the PCB. Pin 1 of the 4A2D (XC6204 series) regulator chip is marked.*

The official schematic for this board specifies the regulator as being the RT9193-33, which supports 300 mA.[1] It is possible that my PCB is a clone using a cheaper regulator chip. My XC6204 series regulator chip is limited to 150 mA. Unless you know the specifics of your unit, it is safest to assume 150 mA is the current limit.

The power performance of the MCU will be examined in a later chapter. But as a starting reference point, the blink program in the device as supplied uses about 30 mA (measured with the +5-volt input supply at 4.97 volts). This measurement includes the small additional current used by the regulator itself.

The datasheet for the STM32F103C8T6 documents the maximum current draw at about 50 mA. This document measurement was obtained with the external clock and

all peripherals enabled, while operating in "run mode" at 72 MHz. Subtracting 50 from your regulator max of 150 leaves you a current budget of about 100 mA from the +3.3-volt regulator. It's always good to know what the limits are!

USB Power/+5V

When powered by a USB cable, the power arrives by the Micro-USB B connector. This +5-volt supply is regulated to the +3.3 volts needed by the MCU. Similarly, at the top right of Figure 3-1, there is a pin labelled "+5V" (with a plus sign), which can be used as a power input. This goes to the same regulator input that the USB connector supplies.

Because of the low current requirements of your MCU, you can also power the unit from a TTL serial adapter. Many USB serial adapters will have a +5-volt line available that can supply your MCU. Check your serial adapter for specifications to be certain.

Be careful *not* to connect a USB cable and supply +5 volts *simultaneously*. Doing so could cause damage to your desktop/laptop through the USB cable. For example, if your +5-volt supply is slightly higher in voltage, you will be injecting current into your desktop USB circuit.

+3.3V Supply

If you have a +3.3-volt power supply, you can leave the +5V inputs *unconnected*. Connect your +3.3-volt power supply directly to the +3.3V input (make sure that the USB cable is *unplugged*). This works because the regulator disables itself when there is no input provided on the 5-volt input.

When supplying power to the +3.3-volt input, you are connecting your power to the VOUT terminal of the regulator shown in Figure 3-3. In this case, there is no 5-volt power flowing into VIN of the regulator. The CE pin is also connected to VIN, but when VIN is unconnected, the CE pin becomes grounded by a capacitor. A low level on CE causes the regulator to shut down its internal subsystems.
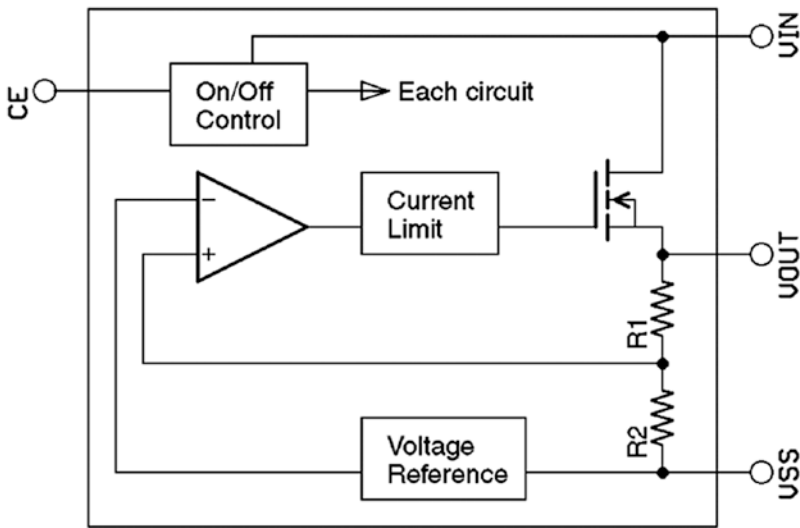
Figure 3-3. *Block diagram of the 5 to 3.3 volt regulator*

There is, however, a small amount of current flow into the regulator's voltage divider. This current will flow from your +3.3 volts to ground, through *internal* resistances R1 and R2 of the regulator. These resistances are high, and the current involved is negligible. But be aware of this when measuring current for ultra-low-power battery applications.

Caution   Do not supply both +5 volts and +3.3 volts at the same time. This could cause damage to the regulator or your desktop when the USB cable is plugged in. Use a single power source.

One Power Source Rule

What I've been leading up to is the general advice to use just *one power source.* I can't stress enough that supplying your PCB with more than one power source can cause damage.

This tends to be obvious with the +3.3-volt and +5-volt supply inputs. What can *easily* be forgotten, however, is the *USB cable.* Consider that you could have power arriving from a USB serial adapter, the ST-Link V2 programmer, or the USB cable. Move slowly when changing your power arrangement, especially when switching from programming the device to your normal power configuration.

Certain applications may require you to use additional supplies; for example, when powering motors or relays. In those cases, you would supply the *external* circuits with the power they need but not the MCU PCB. Only the signals and the ground need to share connections. If this isn't clear, then assume the one power source rule.

Ground

The return side of the power circuit, or the negative side, is known as the ground connection. It is labeled in Figure 3-1 in black. All of these ground connections are electrically connected together. These pins can be used interchangeably.

Reset

The PCB also supplies a button labeled "RESET" and a connection on one side labeled "R." This connection permits an external circuit to reset the MCU if required. Figure 3-4 illustrates the push-button circuit, including the connection going to the MCU.
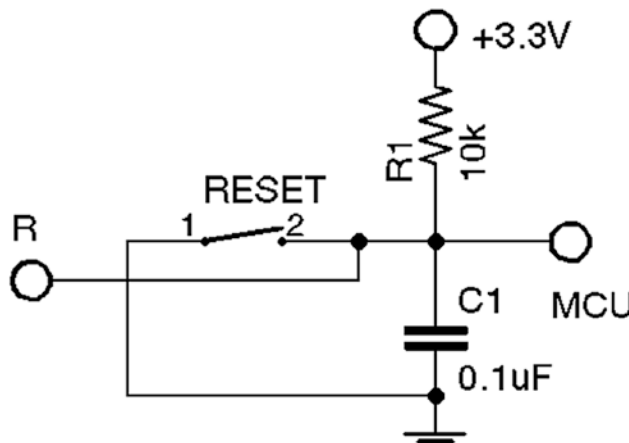
Figure 3-4. *The STM32F103C8T6 Reset circuit. Connection "R" is found on the edge of the PCB.*

Showtime

You've probably already tested your unit, but if you haven't yet then do so now. The safest and easiest way to do this is to use a USB cable with a Micro-USB B connector. Plug your cable into a USB power source, which doesn't have to be a computer.

Once powered, your unit should blink away. If not, then try pressing the Reset button. Also make sure that your boot-0 and boot-1 jumpers are positioned as shown in Figure 3-1 (both jumpers should be positioned to the side labeled "0").

There are two built-in LEDs. The LED on the left indicates that power has been applied (mine was yellow, but yours may differ). The LED at right is activated by GPIO port PC13 under program control (mine was red; again, yours may differ).

Caution   Some have reported having their USB connector break off of the PCB. Be gentle inserting the Micro-USB B cable end.

If you are currently lacking a suitable USB cable, you can try the unit out if you can supply either +5 volts or +3.3 volts to the appropriate connection as discussed. Even a pair of dry cells in series for +3 volts will do (recall that this MCU will function on 2 to 3.3 volts).

Figure 3-5 illustrates the unit's being powered from the +3.3-volt connection at the top of the PCB where the programmer connects. Be careful when using alligator clips, ensuring they don't short to other pins. DuPont wires can be used with greater safety.
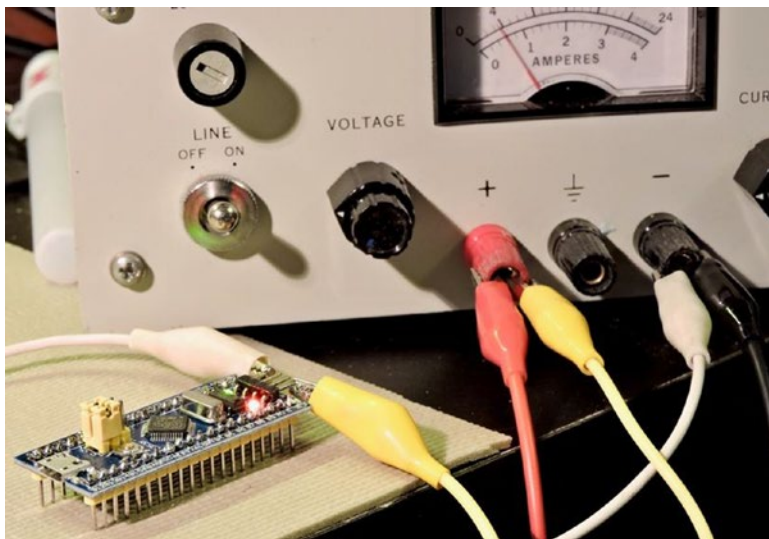
Figure 3-5. *The STM32F108C8T6 blinking and powered by a HP 6284A power supply using the top header strip (+3.3 volts)*

ST-Link V2

The next item to check off our list in this chapter is to hook up and run the `st-info` utility. When you get your programmer, you will likely just get four DuPont wires with female ends. This isn't real convenient but does work if you wire it correctly. If you switch devices to be programmed frequently, you'll want to make a custom cable for the purpose. The programmer hookup diagram is shown in Figure 3-6. It has been reported that different models of the programmer are available using different connections and wiring.
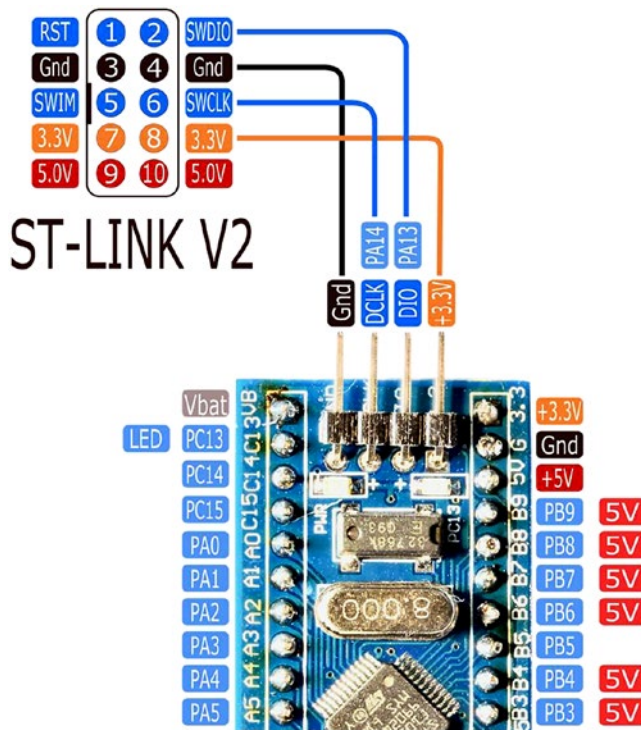
Figure 3-6. *ST-LINK V2 programmer hookup to STM32F103C8T6 device. Check the connections on the device you have, assome ST-Link programmers are different.*

With the programmer hooked up according to Figure 3-6, check your boot-0 and boot-1 jumpers located beside the Reset button. These should appear as they do in Figure 3-1 (with both jumpers close to the side marked "0").

Plug your ST-Link V2 programmer into a USB port or use a USB extension cable. Once you do this, the power LED should immediately light. Also, the PC13 LED should also blink if your unit still has the blink program in it. Figure 3-7 illustrates the setup.
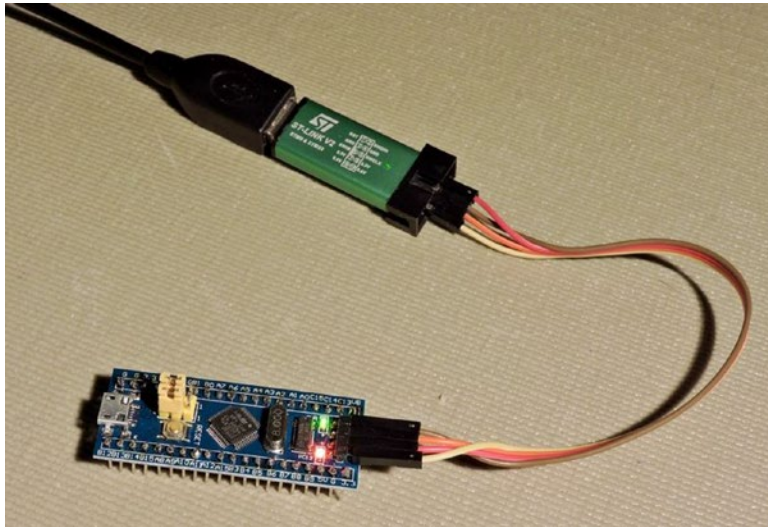
Figure 3-7. *ST-Link V2 programmer using a USB extension cable, hooked up to the STM32F103C8T6 using DuPont wires*

From your desktop, run the st-info command as follows:

```
$ st-info --probe
Found 1 stlink programmers
 serial: 493f6f06483f53564554133f
openocd: "\x49\x3f\x6f\x06\x48\x3f\x53\x56\x45\x54\x13\x3f"
  flash: 131072 (pagesize: 1024)
   sram: 20480
 chipid: 0x0410
  descr: F1 Medium-density device
```

The st-info command should find your ST-Link V2 programmer and the STM32F103C8T6 attached to it. The successful result should be similar to mine shown. Notice that the CPU serial number is reported along with the SRAM (20K). The amount of flash memory reported here is 128K, but you might see 64K instead. It will probably support 128K anyway.

st-flash Utility

Let's now look at how you can use the st-flash utility to read (save), write (program), or erase your STM32 device.

Read STM32

Saving memory content from your device to a file will allow you to restore the original programming, should you need it later. The following example reads from your device's flash memory, starting at address 0x80000000, and saves 0x1000 (4K) worth of data to a file named saved.img. Unless otherwise stated, the C programming 0x prefix convention will be used to indicate hexadecimal numbers in this book:

```
$ st-flash read ./saved.img 0x8000000 0x1000
st-flash 1.3.1-9-gc04df7f-dirty
2017-07-29T09:54:02 INFO src/common.c: Loading device parameters....
2017-07-29T09:54:02 INFO src/common.c: Device connected is: \
    F1 Medium-density device, id 0x20036410
2017-07-29T09:54:02 INFO src/common.c: SRAM size: 0x5000 bytes (20 KiB), \
    Flash: 0x20000 bytes (128 KiB) in pages of 1024 bytes
```

To check the content of the saved image file, use the hexedit utility (you may need to use your package manager to install it on your desktop):

```
$ hexedit saved.img
```

To get help while in the utility, press F1. You can use Control-V to scroll down a page at a time. Use Control-C to exit out to the command line.

Examining the file, you should see hexadecimal content until about offset 0x4EC. From that point on, you may see hexadecimal 0xFF bytes, representing unwritten (erased) flashed memory. If you see nothing but zeros or 0xFF bytes, then something is wrong. Make sure you include the 0x prefix on the address and size arguments of the command.

If you don't see a bunch of 0xFF bytes at the end of the saved image, it may be that you need to save a larger-sized image.

Write Image

Writing flash memory is the reverse of reading, of course. A saved memory image can be "flashed" by use of the write subcommand using st-flash. Note that we omit the size argument for this command. For this example, we write it back to the same address:

```
$ st-flash write ./saved.img 0x8000000
st-flash 1.3.1-9-gc04df7f-dirty
2017-07-29T10:00:39 INFO src/common.c: Loading device parameters....
2017-07-29T10:00:39 INFO src/common.c: Device connected is: \
    F1 Medium-density device, id 0x20036410
2017-07-29T10:00:39 INFO src/common.c: SRAM size: 0x5000 bytes (20 KiB), \
    Flash: 0x20000 bytes (128 KiB) in pages of 1024 bytes
2017-07-29T10:00:39 INFO src/common.c: Ignoring 2868 bytes of 0xff \
    at end of file
2017-07-29T10:00:39 INFO src/common.c: Attempting to write 1228 (0x4cc) \
    bytes to stm32 address: 134217728 (0x8000000)
Flash page at addr: 0x08000400 erased
2017-07-29T10:00:39 INFO src/common.c: Finished erasing 2 pages of 1024 \
    (0x400) bytes
2017-07-29T10:00:39 INFO src/common.c: Starting Flash write for \
    VL/F0/F3 core id
2017-07-29T10:00:39 INFO src/flash_loader.c: Successfully loaded flash \
    loader in sram
  1/1 pages written
2017-07-29T10:00:39 INFO src/common.c: Starting verification of write \
    complete
2017-07-29T10:00:39 INFO src/common.c: Flash written and verified! \
    jolly good!
```

This operation will restore your saved blink image file to the flash memory in your device. It may start to blink immediately. Otherwise, press the Reset button to force a restart.

Erase Flash

There may be times when you want to force a full erasure of the device. Perhaps you want to donate your device to a friend and want to eliminate your last experiment:

```
$ st-flash erase
st-flash 1.3.1-9-gc04df7f-dirty
2017-07-29T10:06:17 INFO src/common.c: Loading device parameters....
2017-07-29T10:06:17 INFO src/common.c: Device connected is: \
    F1 Medium-density device, id 0x20036410
2017-07-29T10:06:17 INFO src/common.c: SRAM size: 0x5000 bytes (20 KiB), \
    Flash: 0x20000 bytes (128 KiB) in pages of 1024 bytes
Mass erasing
```

After this operation completes, your device should be fully erased. It should also stop blinking. For fun, try now to restore the image and reset.

Summary

This chapter provided important information about powering options. This is critical because failures in this area can lead to permanent damage. By now, you have plugged in your unit and verified that it is functional with the included blink program. Then, you confirmed that the programmer and the device to be programmed are both functional using the st-link command. Finally, you learned how to use the st-flash utility to read, write, and erase flash memory on the device.

Bibliography

1. http://wiki.stm32duino.com/images/c/c1/Vcc-gnd.com-STM32F103C8-schematic.pdf

CHAPTER 4

GPIO

In this chapter, you're going to use the libopencm3 library to build a blink program from source code. This example program demonstrates the configuration and use of GPIO (General Purpose Input/Output). The program presented is a slightly modified version of a libopencm3 example program named `miniblink`. It has been modified to provide a different timing so that it will be obvious that your newly flashed code is the one executing. After building and running this program, we'll discuss the GPIO API (Application Programming Interface) that is provided by libopencm3.

Building miniblink

Change to the subdirectory `miniblink` as shown, and type `make`:

```
$ cd ~/stm32f103c8t6/miniblink
$ make
gmake: Nothing to be done for 'all'.
```

If you see the preceding message, it may be because you have already built all of the projects from the top level (there is nothing wrong with that). If, however, you made changes to the source-code files, `make` should automatically detect this and rebuild the affected components. Here, we just want to force the rebuilding of the miniblink project. To do this, type `make clobber` in the project directory, and then `make` afterward, as shown:

```
$ make clobber
rm -f *.o *.d generated.* miniblink.o miniblink.d
rm -f *.elf *.bin *.hex *.srec *.list *.map
$ make
...
```

```
arm-none-eabi-size miniblink.elf
   text    data     bss     dec     hex     filename
    696       0       0     696     2b8     miniblink.elf
arm-none-eabi-objcopy -Obinary miniblink.elf miniblink.bin
```

When you do this, you will see a few long command lines executed to compile and link your executable named `miniblink.elf`. To flash your device, however, we also need an image file. The last step of the build shows how the ARM-specific `objcopy` utility is used to convert `miniblink.elf` into the image file `miniblink.bin`.

Just prior to the last step, however, you can see that the ARM-specific `size` command has dumped out the sizes of the data and text sections of your program. Our miniblink program consists only of 696 bytes of flash (section text) and uses no allocated SRAM (section data). While this is accurate, there is still SRAM being used for a call stack.

Flashing miniblink

Using the `make` framework again, we can now flash your device with the new program image. Hook up your ST-Link V2 programmer, check the jumpers, and execute the following:

```
$ make flash
/usr/local/bin/st-flash write miniblink.bin 0x8000000
st-flash 1.3.1-9-gc04df7f-dirty
2017-07-30T12:57:56 INFO src/common.c: Loading device parameters....
2017-07-30T12:57:56 INFO src/common.c: Device connected is: \
    F1 Medium-density device, id 0x20036410
2017-07-30T12:57:56 INFO src/common.c: SRAM size: 0x5000 bytes (20 KiB),\
    Flash: 0x20000 bytes (128 KiB) in pages of 1024 bytes
2017-07-30T12:57:56 INFO src/common.c: Attempting to write 696 (0x2b8) \
    bytes to stm32 address: 134217728 (0x8000000)
Flash page at addr: 0x08000000 erased
2017-07-30T12:57:56 INFO src/common.c: Finished erasing 1 pages of \
    1024 (0x400) bytes
...
2017-07-30T12:57:57 INFO src/common.c: Flash written and verified! \
    jolly good!
```

Once this is done, your device should automatically reset and start the flashed miniblink program. With the modified time constants used, you should see it now blinking frequently, with a mostly-on 70/30 duty cycle. Your supplied device blink program likely used a slower 50/50 duty cycle instead. If your blink pattern varies somewhat from what is described, don't worry. The important point is that you've flashed and run a *different* program.

This program does not use a crystal-controlled CPU clock. It uses the internal RC clock (resistor/capacitor clock). For this reason, your unit may flash quite a bit faster or slower than someone else's unit.

miniblink.c Source Code

Let's now examine the source code for the miniblink program you just ran. If not still in the subdirectory `miniblink`, change to there now:

```
$ cd ~/stm32f103c8t6/miniblink
```

Within this subdirectory, you should find the source program file `miniblink.c`. Listing 4-1 illustrates the program without comment boilerplate:

Listing 4-1. Listing of miniblink.c

```
0019: #include <libopencm3/stm32/rcc.h>
0020: #include <libopencm3/stm32/gpio.h>
0021:
0022: static void
0023: gpio_setup(void) {
0024:
0025:    /* Enable GPIOC clock. */
0026:    rcc_periph_clock_enable(RCC_GPIOC);
0027:
0028:    /* Set GPIO8 (in GPIO port C) to 'output push-pull'. */
0029:    gpio_set_mode(GPIOC,GPIO_MODE_OUTPUT_2_MHZ,
0030:                  GPIO_CNF_OUTPUT_PUSHPULL,GPIO13);
0031: }
0032:
```

```
0033: int
0034: main(void) {
0035:    int i;
0036:
0037:    gpio_setup();
0038:
0039:    for (;;) {
0040:        gpio_clear(GPIOC,GPIO13);       /* LED on */
0041:        for (i = 0; i < 1500000; i++)   /* Wait a bit. */
0042:            __asm__("nop");
0043:
0044:        gpio_set(GPIOC,GPIO13);         /* LED off */
0045:        for (i = 0; i < 500000; i++)    /* Wait a bit. */
0046:            __asm__("nop");
0047:    }
0048:
0049:    return 0;
0050: }
```

Note   The line numbers appearing at the left in the listings are not part of the source file. These are used for ease of reference only.

The structure of the program is rather simple. It consists of the following:

1. A main program function declared in lines 33–50. Note that unlike a POSIX program, there are no argc or argv arguments to function main.

2. Within the main program, function gpio_setup() is called to perform some initialization.

3. Lines 39–47 form an infinite loop, where an LED is turned on and off. Note that the return statement in line 49 is never executed and is provided only to keep the compiler from complaining.

Even in this simple program there is much to discuss. As we will see later, this example program runs at a default CPU frequency since none is defined. This will be explored later.

Let's drill down on the simple things first. Figure 4-1 illustrates how the LED that we're flashing is attached to the MCU on the Blue Pill PCB. In this schematic view, we see that power enters the LED from the +3.3-volt supply through limiting resistor R1. To complete the circuit, the GPIO PC13 must connect the LED to ground to allow the current to flow. This is why the comment on line 40 says that the LED is being turned on, even though the function call is gpio_clear(). Line 44 uses gpio_set() to turn the LED off. This *inverted logic* is used simply because of the way the LED is wired.
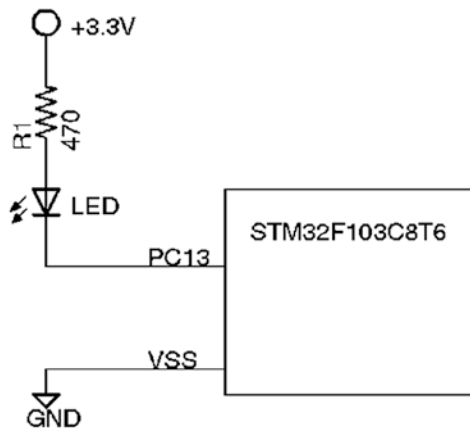
Figure 4-1. *LED connected to PC13 on the Blue Pill PCB*

Look again at these function calls:

```
gpio_clear(GPIOC,GPIO13);     /* LED on */
...
gpio_set(GPIOC,GPIO13);       /* LED off */
```

Notice that these two calls require two arguments, as follows:

1. A GPIO *port* name

2. A GPIO *pin* number

If you are used to the Arduino environment, you are used to using something like the following:

```
int ledPin = 13;  // LED on digital pin 13

digitalWrite(ledPin,HIGH);
...
digitalWrite(ledPin,LOW);
```

43

In the non-Arduino world, you generally work directly with a port and a pin. Within the libopencm3 library, you specify whether you are clearing or setting a bit based upon the function name (gpio_clear() or gpio_set()). You can also toggle a bit with the use of gpio_toggle(). Finally, it is possible to read and write the full set of pins by port alone, using gpio_port_read() and gpio_port_write() respectively.

GPIO API

This is a good place to discuss the libopencm3 functions that are available for GPIO use. The first thing you need to do is include the appropriate header files, as follows:

```
#include <libopencm3/stm32/rcc.h>
#include <libopencm3/stm32/gpio.h>
```

The rcc.h file is needed for definitions so as to enable the GPIO clock. The gpio.h file is necessary for the remainder:

```
void gpio_set(uint32_t gpioport, uint16_t gpios);
void gpio_clear(uint32_t gpioport, uint16_t gpios);
uint16_t gpio_get(uint32_t gpioport, uint16_t gpios);
void gpio_toggle(uint32_t gpioport, uint16_t gpios);
uint16_t gpio_port_read(uint32_t gpioport);
void gpio_port_write(uint32_t gpioport, uint16_t data);
void gpio_port_config_lock(uint32_t gpioport, uint16_t gpios);
```

In all of the preceding functions, the argument gpioport can be one of the macros from Table 4-1 (on other STM32 platforms, there can be additional ports). Only *one port* can be specified at a time.

Table 4-1. *libopencm3 GPIO Macros for STM32F103C8T6*

Port Macro	Description
GPIOA	GPIO port A
GPIOB	GPIO port B
GPIOC	GPIO port C

In the libopencm3 GPIO functions, one or more GPIO bits may be set or cleared at once. Table 4-2 lists the macro names supported. Note also the macro named GPIO_ALL.

Table 4-2. *libopencm3 GPIO pin designation macros*

Pin Macro	Definition	Description
GPIO0	(1 << 0)	Bit 0
GPIO1	(1 << 1)	Bit 1
GPIO2	(1 << 2)	Bit 2
GPIO3	(1 << 3)	Bit 3
GPIO4	(1 << 4)	Bit 4
GPIO5	(1 << 5)	Bit 5
GPIO6	(1 << 6)	Bit 6
GPIO7	(1 << 7)	Bit 7
GPIO8	(1 << 8)	Bit 8
GPIO9	(1 << 9)	Bit 9
GPIO10	(1 << 10)	Bit 10
GPIO11	(1 << 11)	Bit 11
GPIO12	(1 << 12)	Bit 12
GPIO13	(1 << 13)	Bit 13
GPIO14	(1 << 14)	Bit 14
GPIO15	(1 << 15)	Bit 15
GPIO_ALL	0xffff	All bits 0 through 15

An example of GPIO_ALL might be the following:

```
gpio_clear(PORTB,GPIO_ALL); // clear all PORTB pins
```

A special feature of the STM32 series, which libopencm3 supports, is the ability to lock a GPIO I/O definition, as follows:

```
void gpio_port_config_lock(uint32_t gpioport, uint16_t gpios);
```

After calling `gpio_port_config_lock()` on a port for the selected GPIO pins, the I/O configuration is frozen until the next system reset. This can be helpful in safety-critical systems where you don't want an errant program to change these. When a selected GPIO is made an input or an output, it is guaranteed to remain so.

GPIO Configuration

Let's now examine how the GPIO was set up in function `gpio_setup()`. Line 26 of Listing 4-1 has the following curious call:

```
rcc_periph_clock_enable(RCC_GPIOC);
```

You will discover throughout this book that the STM32 series is very configurable. This includes the underlying clocks needed for the various GPIO ports and peripherals. The shown libopencm3 function is used to turn on the system clock for GPIO port C. If this clock were not enabled, GPIO port C wouldn't function. Sometimes the affected software will have operations ignored (visible result), while in other situations the system can seize up. Consequently, this is one of those critical "ducks" that needs to be "in a row."

The reason that clocks are disabled at all is to save on power consumption. This is important for battery conservation.

Tip   If your peripheral or GPIO is not functioning, check that you have enabled the necessary clock(s).

The next call made is to `gpio_set_mode()` in line 29:

```
 gpio_set_mode(
    GPIOC,                      // Table 4-1
    GPIO_MODE_OUTPUT_2_MHZ,     // Table 4-3
    GPIO_CNF_OUTPUT_PUSHPULL,   // Table 4-4
    GPIO13                      // Table 4-2
 );
```

This function requires four arguments. The first argument specifies the affected GPIO port (Table 4-1). The fourth argument specifies the GPIO pins affected (Table 4-2). The third argument's macro values are listed in Table 4-3 and define the general mode of the GPIO port.

Table 4-3. *GPIO Mode Definitions*

Mode Macro Name	Value	Description
GPIO_MODE_INPUT	0x00	Input mode
GPIO_MODE_OUTPUT_2_MHZ	0x02	Output mode, at 2 MHz
GPIO_MODE_OUTPUT_10_MHZ	0x01	Output mode, at 10 MHz
GPIO_MODE_OUTPUT_50_MHZ	0x03	Output mode, at 50 MHz

The macro GPIO_MODE_INPUT defines the GPIO pin as an input, as you would expect. But there are three output mode macros listed.

Each output selection affects how quickly each output pin responds to a change. In our example program, the 2 MHz option was selected. This was chosen because the speed of an LED signal change is not going to be noticed by human eyes. By choosing 2 MHz, power is saved and EMI (electromagnetic interference) is reduced.

The third argument further specializes how the port should be configured. Table 4-4 lists the macro names provided.

Table 4-4. *I/O Configuration Specializing Macros*

Specialization Macro Name	Value	Description
GPIO_CNF_INPUT_ANALOG	0x00	Analog input mode
GPIO_CNF_INPUT_FLOAT	0x01	Digital input, floating (default)
GPIO_CNF_INPUT_PULL_UPDOWN	0x02	Digital input, pull up and down
GPIO_CNF_OUTPUT_PUSHPULL	0x00	Digital output, push/pull
GPIO_CNF_OUTPUT_OPENDRAIN	0x01	Digital output, open drain
GPIO_CNF_OUTPUT_ALTFN_PUSHPULL	0x02	Alternate function output, push/pull
GPIO_CNF_OUTPUT_ALTFN_OPENDRAIN	0x03	Alternate function output, open drain

Input Ports

The macro names including INPUT only apply when the second argument implies an input port. We see from Table 4-4 that inputs can be specialized three different ways:

- Analog

- Digital, floating input

- Digital, pull up and down

To make greater sense of the GPIO input and its configuration, examine the simplified Figure 4-2.
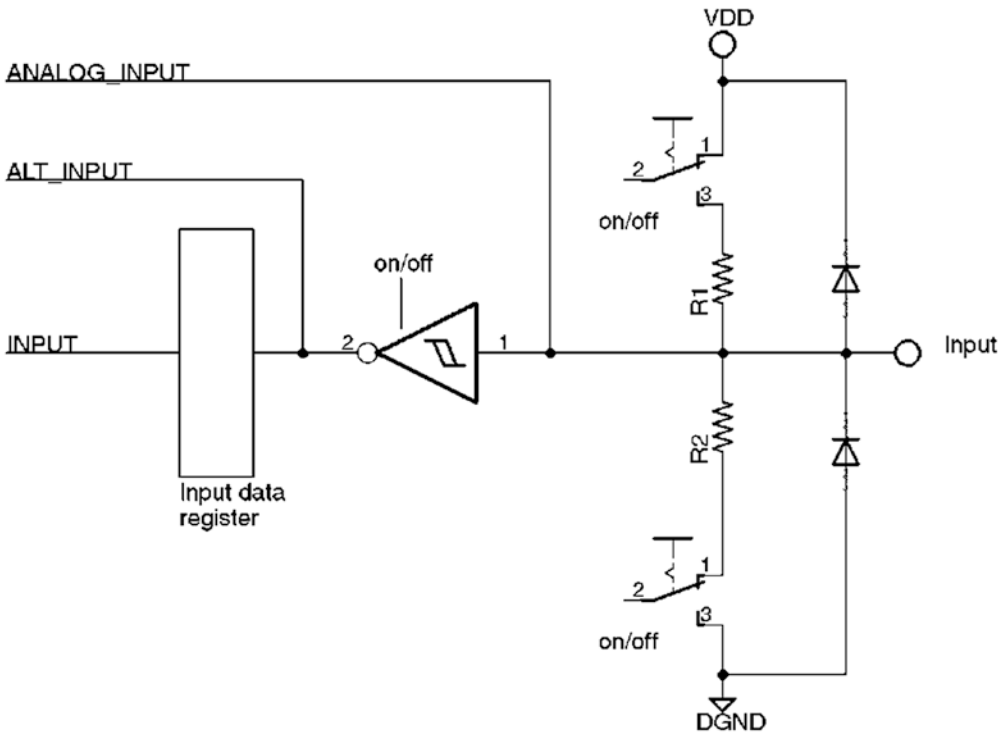
Figure 4-2. *Basic structure of the GPIO input*

The MCU receiving side is on the left side, while the external input comes in from the right. There are two protection diodes attached, which normally only come into play if the static voltage goes negative or exceeds the supply.

When the input port is configured as an *analog* input, the switches connected to resistors R1 and R2 are switched off. This is to avoid pulling the analog signal up or

down. With the resistors disconnected, the analog input is routed to the line labeled "Analog Input" with no further signal effect for the ADC (analog-to-digital conversion) peripheral. The Schmitt trigger is also disabled to save on power consumption.

When the input port is configured for *digital* input, resistors R1 or R2 are in operation unless you select the "float" option GPIO_CNF_INPUT_FLOAT. For both digital input modes, the Schmitt trigger is enabled to provide a cleaner signal with hystersis. The output of the Schmitt trigger then goes to the "Alternate Function Input" and to the input data (GPIO) register. More will be said about alternate functions later, but the simple answer is that an input can act as a GPIO input or as a peripheral input.

The 5-volt-tolerant inputs are identical to the diagram shown in Figure 4-2, except that the high side protective diode allows the voltage to rise above 3.3 volts to at least +5 volts.

Note   When configuring a peripheral output, be sure to use one of the alternate function macros. Otherwise, only GPIO signals will be configured.

Output Ports

When the GPIO port is configured for output, you have four specializations to choose from:

- GPIO push/pull

- GPIO open drain

- Alternate function push/pull

- Alternate function open drain

For GPIO operation, you always choose the *non-alternate* function modes. For peripheral use like the USART, you choose from the alternate function modes instead. A common mistake is to configure for GPIO use, like GPIO_CNF_OUTPUT_PUSHPULL for the TX output of the USART. The correct macro is GPIO_CNF_OUTPUT_ALTFN_PUSHPULL for the peripheral. If you're not seeing peripheral output, ask yourself if you chose from the one of the *alternate function* values.

Figure 4-3 illustrates the block diagram for GPIO outputs. For 5-volt-tolerant outputs (like the inputs), the only change to the circuit is that the high side protective diode is capable of accepting voltages as high as +5 volts. For non-5-volt-tolerant ports, the high side protective diode can only rise to +3.3 volts (actually, it can rise to one diode drop above 3.3 volts).

The "output control" circuit determines if it is driving the P-MOS and N-MOS transistors (in push/pull mode) or just the N-MOS (in open-drain mode). In open-drain mode, the P-MOS transistor is always kept off. Only when you write a zero to the output will the N-MOS transistor turn on and pull the output pin low. Writing a 1-bit to an open-drain port effectively disconnects the port since both transistors are put into the "off" state.

The weak input resistors shown in Figure 4-2 are disabled in output mode. For this reason, they were omitted from Figure 4-3.

The output data bits are selected from either the output (GPIO) data register or the *alternate function* source. GPIO outputs go to the output data register, which can be written as an entire word or as individual bits. The bit set/reset register permits individual GPIO bits to be altered as if they were one atomic operation. In other words, an interrupt cannot occur in the middle of an "and/or" operation on a bit.

Because GPIO output data is captured in the output data register, it is possible to read back what the current output settings are. This doesn't work for alternate function configurations, however.
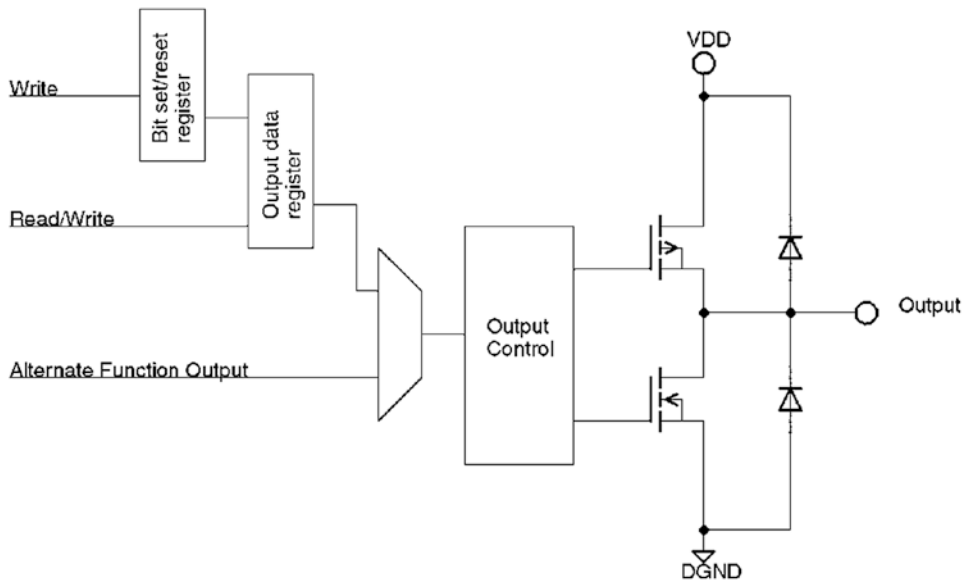
Figure 4-3. *The output GPIO driver circuit*

When the output is configured for a peripheral like the USART, the data comes from the peripheral through the alternate function output line. Seeing how the data is steered in Figure 4-3 should emphasize the fact that you must configure the port for GPIO *or* alternate functions. I am harping on this so that you won't waste your time having to debug this kind of problem.

Ducks in a Row

While the origin of the saying "to have one's ducks in a row" is unclear, the one possibility that I like refers to the fairground amusement of shooting at a row of mechanical ducks. This arrangement makes it easier for the shooter to get them all and win the prize.

Peripherals on the STM32 platform are highly configurable, which also leaves more than the usual opportunity for mistakes. Consequently, I'll refer often to this idea of getting your ducks in a row, as a shorthand recipe for success. When your peripheral configuration is not working as expected, review the ducks-in-a-row list.

Often, the problem is an omission or the use of an incorrect macro that failed to raise a compiler warning. Sequence is also often important—you need to enable a clock before configuring a device that needs that clock, for example.

GPIO Inputs

When configuring GPIO *input* pins, use the following procedure to configure it. This applies to *GPIO* inputs only—not a peripheral input, like the USART. Peripherals require other considerations, especially if *alternate pin configurations* are involved (they will be covered later in the book).

1. *Enable the GPIO port clock*. For example, if the GPIO pin is on port C, then enable the clock with a call to `rcc_periph_clock_enable(RCC_GPIOC)`. You *must* enable each *port* used individually, using the `RCC_GPIOx` macros.

2. Set the mode of the input pin with `gpio_set_mode()`, specifying the port in argument one, and the `GPIO_MODE_INPUT` macro in argument two.

3. Within the gpio_set_mode() call, choose the appropriate specialization macro GPIO_CNF_INPUT_ANALOG, GPIO_CNF_INPUT_ FLOAT, or GPIO_INPUT_PULL_UPDOWN as appropriate.

4. Finally, specify in the last argument in the gpio_set_mode() call all pin numbers that apply. These are or-ed together, as in GPIO12|GPIO15, for example.

Digital Output, Push/Pull

Normally, digital outputs are configured for push/pull mode. This ducks-in-a-row advice is provided for this normal case:

1. *Enable the GPIO port clock.* For example, if the GPIO pin is on port B, then enable the clock with a call to rcc_periph_clock_ enable(RCC_GPIOB). You must enable each *port* used individually, using the RCC_GPIOx macros.

2. Set the mode of the output pin with gpio_set_mode(), specifying the port in argument one and one of the GPIO_MODE_OUTPUT_*_MHZ macros in argument two. For non-critical signal rates, choose the lowest value GPIO_MODE_OUTPUT_2_MHZ to save power and to lower EMI.

3. Specify GPIO_CNF_OUTPUT_PUSHPULL in argument three in the call to gpio_set_mode(). Do not use any of the ALTFN macros for GPIO use (those are for peripheral use only).

4. Finally, specify in the last argument in the gpio_set_mode() call all pin numbers that apply. These are or-ed together, as in GPIO12|GPIO15, for example.

Digital Output, Open Drain

When working with a *bus*, where more than one transistor may be used to pull down a voltage, an open-drain output may be required. Examples are found in I2C or CAN bus communications. The following procedure is recommended for *GPIO* open-drain outputs *only* (do not use this procedure for peripherals):

1. *Enable the GPIO port clock*. For example, if the GPIO pin is on port B, then enable the clock with a call to `rcc_periph_clock_enable(RCC_GPIOB)`. You must enable each *port* used individually, using the `RCC_GPIOx` macros.

2. Set the mode of the output pin with `gpio_set_mode()`, specifying the port in argument one, and one of the `GPIO_MODE_OUTPUT_*_MHZ` macros in argument two. For non-critical signal rates, choose the lowest value `GPIO_MODE_OUTPUT_2_MHZ` to save power and to lower EMI.

3. Specify `GPIO_CNF_OUTPUT_OPENDRAIN` in argument three in the call to `gpio_set_mode()`. Do not use any of the `ALTFN` macros for GPIO use (those are for peripheral use).

4. Finally, specify in the last argument in the `gpio_set_mode()` call all pin numbers that apply (one or more). These are or-ed together, as in `GPIO12|GPIO15`, for example.

GPIO Characteristics

This is a good place to summarize the capabilities of the STM32 GPIO pins. Many are 5-volt tolerant as inputs, while a few others are current limited for output. Using the STM32 documentation convention, ports are often referenced as `PB5,` for example, to refer to GPIO port B pin GPIO5. I'll be using this convention throughout this book. Table 4-5 summarizes these important GPIO characteristics as they apply to the Blue Pill device.

Table 4-5. *GPIO Capabilities: Except Where Noted, All GPIO Pins Can Source or Sink a Maximum of 25 mA of Current*

Pin	GPIO_PORTA			GPIO_PORTB			GPIO_PORTC		
	3V/5V	Reset	Alt	3V/5V	Reset	Alt	3V/5V	Reset	Alt
GPIO0	3V	PA0	Yes	3V	PB0	Yes			
GPIO1	3V	PA1	Yes	3V	PB1	Yes			
GPIO2	3V	PA2	Yes	5V	PB2/BOOT1	No			
GPIO3	3V	PA3	Yes	5V	JTDO	Yes			
GPIO4	3V	PA4	Yes	5V	JNTRST	Yes			
GPIO5	3V	PA5	Yes	3V	PB5	Yes			
GPIO6	3V	PA6	Yes	5V	PB6	Yes			
GPIO7	3V	PA7	Yes	5V	PB7	Yes			
GPIO8	5V	PA8	No	5V	PB8	Yes			
GPIO9	5V	PA9	No	5V	PB9	Yes			
GPIO10	5V	PA10	No	5V	PB10	Yes			
GPIO11	5V	PA11	No	5V	PB11	Yes			
GPIO12	5V	PA12	No	5V	PB12	No			
GPIO13	5V	JTMS/SWDIO	Yes	5V	PB13	No	3V	3 mA @ 2 MHz	Yes
GPIO14	5V	JTCK/SWCLK	Yes	5V	PB14	No	3V	3 mA @ 2 MHz	Yes
GPIO15	5V	JTDI	Yes	5V	PB15	No	3V	3 mA @ 2 MHz	Yes

The column ALT in Table 4-5 indicates where alternate functions can apply. Input GPIOs marked with "5V" can safely tolerate a 5-volt signal, whereas the others marked "3V" can only accept signals up to +3.3 volts. The column labeled Reset indicates the state of the GPIO configuration after an MCU reset has occurred.

GPIO pins PC13, PC14, and PC15 are current limited. These can *sink* a maximum of 3 mA and should never be used to *source* a current. Additionally, the documentation indicates that these should never be configured for operations of more than 2 MHz when configured as outputs.

Input Voltage Thresholds

Given that the STM32F103C8T6 can operate over a range of voltages, the GPIO input-threshold voltages follow a formula. Table 4-6 documents what you can expect for the Blue Pill device, operating at +3.3 volts.

Table 4-6. *Input-Voltage Thresholds Based Upon V_{DD} = +3.3volts*

Symbol	Description	Range
V_{IL}	Standard low-input voltage	0 to 1.164 volts
	5-volt-tolerant inputs	0 to 1.166 volts
V_{IH}	High-input voltage	1.155 to 3.3/5.0 volts

You may have noticed that there is a small overlap between the high end of the V_{IL} and the low end of the V_{IH} range. The STM32 documentation indicates that there is about 200 mV of hysterisis between these input states.

Output-Voltage Thresholds

The output GPIO thresholds are documented in Table 4-7, based upon the Blue Pill device operating at +3.3 volts. Note that the ranges degrade as current increases.

Table 4-7. *GPIO Output-Voltage Levels with Current <= 20 mA*

Symbol	Description	Range
V_{OL}	Output voltage low	0.4 to 1.3 volts
V_{OH}	Output voltage high	2 to 3.3 volts

Programmed Delays

Returning now to the program illustrated in Listing 4-1, let's examine the timing aspect of that program, repeated here for convenience:

```
0039:   for (;;) {
0040:       gpio_clear(GPIOC,GPIO13);       /* LED on */
0041:       for (i = 0; i < 1500000; i++)  /* Wait a bit. */
0042:           __asm__("nop");
0043:
0044:       gpio_set(GPIOC,GPIO13);         /* LED off */
0045:       for (i = 0; i < 500000; i++)   /* Wait a bit. */
0046:           __asm__("nop");
0047:   }
```

The first thing to notice about this segment is that the loop counts differ: 1,500,000 in line 41 and 500,000 in line 45. This causes the LED to remain on 75 percent of the time and turn off for 25 percent.

The __asm__("nop") statement forces the compiler to emit the ARM assembler instruction nop as the body of both loops. Why is this necessary? Why not code an empty loop like the following?

```
0041:       for (i = 0; i < 1500000; i++)     /* Wait a bit. */
0042:           ;  /* empty loop */
```

The problem with an empty loop is that the compiler may optimize it away. Compiler optimization is always being improved, and this type of construct could be seen as redundant and be removed from the compiled result. This feature is also sensitive to the optimize options used for the compile. This __asm__ trick is one way to force the compiler to always produce the loop code and perform the nop (no operation) instruction.

The Problem with Programmed Delay

The good thing about programmed delays is that they are easy to code. But, beyond that, there are problems:

- How many iterations do I need for a timed delay?
- Poor source code portability:
    - the delay will vary for different platforms
    - the delay will vary by CPU clock rate
    - the delay will vary by different execution contexts
- Wastes CPU, which could be used by other tasks in a multi-tasking environment
- The delays are unreliable when preemptive multi-tasking is used

The first problem is the difficulty of computing the number of iterations needed to achieve a delay. This loop count depends upon several factors, as follows:

- The CPU clock rate
- The instruction cycle times used
- Single or multi-tasking environment

In the miniblink program, there was no CPU clock rate established. Consequently, this code is at the mercy of the default used. By experiment, loop counts that "seem to work" can be derived. But if you run the same loops from SRAM instead of flash, the delays will be shorter. This is because there are no wait cycles necessary to fetch the instruction words from SRAM. Fetching instructions from flash, on the other hand, may involve wait cycles, depending upon the CPU clock rate chosen.

In a multi-tasking environment, like FreeRTOS, programmed delays are a poor choice. One reason is because you don't know how much time is consumed by the other tasks.

Finally, programmed delays are not portable to other platforms. Perhaps the source code will be reused on an STM32F4 device, where the execution efficiency is different. The code will need manual intervention to correct the timing deficiency.

All of these reasons are why FreeRTOS provides an API for timing and delay. This will be examined later when we apply FreeRTOS in our demo programs.

Summary

This chapter has necessarily covered a lot of ground, even though we're just getting started. You've exercised the `st-flash` utility and programmed your device with the miniblink program, which was a different blink program than the one supplied with your unit.

More interestingly, the libopencm3 GPIO API was discussed, and the miniblink program was examined in detail. This explained GPIO configuration and operations. Finally, the problems of programmed delays were discussed.

EXERCISES

1. What GPIO port does the built-in LED on the Blue Pill PCB use? Specify the libopencm3 macro name for the port.

2. What GPIO pin does the built-in LED on the Blue Pill PCB use? Specify the libopencm3 macro name.

3. What level is required to turn the built-in LED on for the Blue Pill PCB?

4. What are two factors affecting the chosen loop count in a programmed delay in non-multi-tasking environments?

5. Why are programmed delays not used in a multi-tasking environment?

6. What three factors affect instruction timing?

7. What are the three modes of an input GPIO port?

8. Do the weak pull-up and pull-down resistors participate in an analog input?

9. When is the Schmitt trigger enabled for input ports?

10. Do the weak pull-up and pull-down resistors participate for output GPIO ports?

11. When configuring a USART TX (transmit) output for push/pull operation, which specialization macro should be used?

12. When configuring a pin for LED use, which GPIO mode macro is preferred for low EMI?

CHAPTER 5

FreeRTOS

Early in this book, we transition to the use of FreeRTOS. Doing so offers a number of advantages, mainly because the programming becomes much simpler and offers greater reliability as a result.

Not all platforms are capable of supporting an RTOS (real-time operating system). Each task within a multi-tasking system requires some stack space to be allocated where variables and return addresses of function calls are stored. There simply isn't much RAM on an ATmega328, for example, with only 2K. The STM32F103C8T6, on the other hand, has 20K of SRAM available to divide among a reasonable complement of tasks.

This chapter introduces FreeRTOS, which is open sourced and available for free. The FreeRTOS source code is licensed under GPL License 2. There is a special provision to allow you to distribute your linked product without requiring the distribution of your own proprietary source code. Look for the text file named LICENSE for the details.

FreeRTOS Facilities

What makes an RTOS desirable? What does it provide? Let's examine some major categories of services found in FreeRTOS:

- Multi-tasking and scheduling

- Message queues

- Semaphores and mutexes

- Timers

- Event groups

Tasking

In the Arduino environment, everything runs as a single task, with a single stack for variables and return addresses. This style of programming requires you to run a loop polling each event to be serviced. With every iteration, you may need to poll the temperature sensor and then invoke another routine as part of the loop to broadcast that result.

With FreeRTOS (an RTOS), logical functions are placed into separate tasks that run independently. One task might be responsible for reading and computing the current temperature. Another task could be responsible for broadcasting that last computed temperature. In effect, it becomes a pair of programs running at the same time.

For very simple applications, this overhead of task scheduling might be seen as overkill. However, as complexity increases, the advantages of partitioning the problem into tasks become much more pronounced.

FreeRTOS is very flexible. It provides two types of task scheduling:

- Preemptive multi-tasking

- Cooperative multi-tasking (coroutines)

With preemptive multi-tasking, a task runs until it runs out of its time slice, or becomes blocked, or yields control explicitly. The task scheduler manages which task is run next, taking priorities into account. This is the type of multi-tasking that will be used within this book's projects.

Another form of multi-tasking is coroutines. The difference is that the current task runs until it gives up control. There is no time slice or timeout. If no function call would block (like a mutex), then a coroutine must call a yield function in order to hand control over to another task. The task scheduler then decides which task to pass control to next. This form of scheduling is desirable for safety-critical applications needing strict control of CPU time.

Message Queues

As soon as you adopt multi-tasking, you inherit a communication problem. Using our temperature-reading example, how does the temperature-reading task safely communicate the value to the temperature-broadcasting task? If the temperature is stored as four bytes, how do you pass that value without interruption? Preemptive multi-tasking means that copying four bytes of data to another location might get interrupted partway through.

A crude way to solve this would be to inhibit interrupts while copying your temperature to a location used by the broadcast task. But this approach could be intolerable if you have frequently occurring interrupts. The problem worsens when the objects to be copied increase in size.

The message-queue facility within FreeRTOS provides a task-safe way to communicate a complete message. The message queue guarantees that only complete messages will be received. Additionally, it limits the length of the queue so that a sending task can't use up all the memory. By using a predetermined queue length, the task adding messages becomes blocked until space is available. When a task becomes blocked, the task scheduler automatically switches to another task that is ready to run, which may remove messages from that same queue. The fixed length gives the message queue a form of flow control.

Semaphores and Mutexes

Within the implementation of a queue, there is a mutex operation at work. The process of adding a message may require several instructions to complete. Yet, in a preemptive multi-tasking system, it is possible for a message to be half added before being interrupted to execute another task.

Within FreeRTOS, the queue is designed to have messages added in an atomic manner. To accomplish this, some sort of mutex device is used behind the scenes. The mutex is an all-or-nothing device. You either have the lock or don't.

Similar to mutexes, there are semaphores. In some situations where you might want to limit a certain number of concurrent requests, for example, a semaphore can manage that in an atomic manner. It might allow a maximum value of three, for example. Then, up to three "take" requests will succeed. Additional "take" requests will block until one or more "give" requests have been made to give back the resource.

Timers

Timers are important for many applications, including the blink variety of program. When you have multiple tasks consuming CPU time, a delay routine is not only unreliable, but it also robs other tasks of CPU time that could have been used more productively.

Within an RTOS system, there is usually a "systick" interrupt that helps with time management. This systick interrupt not only tracks the current number of "ticks" issued so far but is also used by the task scheduler to switch tasks.

Within FreeRTOS, you can choose to delay execution by a specified number of ticks. This works by noting the current "tick time" and yielding to another task until the required tick time has arrived. In this way, the delay precision is limited only to the tick interval configured. It also permits other tasks to do real work until the right time arrives.

FreeRTOS also has the facility of software timers that can be created. Only when the timer expires is your function callback executed. This approach is memory frugal because all timers will make use of the same stack.

Event Groups

One problem that often occurs is that a task may need to monitor multiple queues at once. For example, a task might need to block until a message arrives from either of two different queues. FreeRTOS provides for the creation of "queue sets." This allows a task to block until a message from any of the queues in the set has a message.

What about user-defined events? Event groups can be created to allow binary bits to represent an event. Once established, the FreeRTOS API permits a task to wait until a specific combination of events occurs. Events can be triggered from normal task code or from within an ISR (interrupt service routine).

The blinky2 Program

Change to the blinky2 demo directory:

```
$ cd ~/stm32f103c8t6/rtos/blinky2
```

This example program uses the FreeRTOS API to implement a blink program in an RTOS environment. Listing 5-1 illustrates the top of the source code file main.c. From this listing, notice the include files used:

- FreeRTOS.h

- task.h

- libopencm3/stm32/rcc.h

- libopencm3/stm32/gpio.h

You have seen the libopencm3 header files before. The `task.h` header file defines macros and functions related to the creation of tasks. Finally, there is `FreeRTOS.h`, which every project needs in order to customize and configure FreeRTOS. We'll examine that after we finish with `main.c`.

The program `main.c` also defines the function prototype for the function named `vApplicationStackOverflowHook()` in lines 11–13 of Listing 5-1. FreeRTOS does not provide a function prototype for it, so we must provide it here to avoid having the compiler complain about it.

Listing 5-1. The Top of stm32/rtos/blinky2/main.c Source Code

```
0001: /* Simple LED task demo, using timed delays:
0002:  *
0003:  * The LED on PC13 is toggled in task1.
0004:  */
0005: #include "FreeRTOS.h"
0006: #include "task.h"
0007:
0008: #include <libopencm3/stm32/rcc.h>
0009: #include <libopencm3/stm32/gpio.h>
0010:
0011: extern void vApplicationStackOverflowHook(
0012:    xTaskHandle *pxTask,
0013:    signed portCHAR *pcTaskName);
```

Listing 5-2 lists the definition of the `vApplicationStackOverflowHook()` optional function. This function could have been left out of the program without causing a problem. It is provided here to illustrate how you would define it, if you wanted it.

Listing 5-2. blinky2/main.c, Function vApplicationStackOverflowHook()

```
0017: void
0018: vApplicationStackOverflowHook(
0019:    xTaskHandle *pxTask __attribute((unused)),
0020:    signed portCHAR *pcTaskName __attribute((unused))
0021: ) {
0022:    for(;;);    // Loop forever here..
0023: }
```

If the function is defined, FreeRTOS will invoke it when it detects that it has overrun a stack limit. This allows the application designer to decide what should be done about it. You might, for example, want to flash a special red LED to indicate program failure.

Listing 5-3 illustrates the task that performs the LED blinking. It accepts a void * argument, which is unused in this example. The __attribute((unused)) is a gcc attribute to indicate to the compiler that the argument args is *unused*, and it prevents warnings about it.

Listing 5-3. blinky2/main.c, Function task1()

```
0025: static void
0026: task1(void *args __attribute((unused))) {
0027:
0028:    for (;;) {
0029:        gpio_toggle(GPIOC,GPIO13);
0030:        vTaskDelay(pdMS_TO_TICKS(500));
0031:    }
0032: }
```

The body of the function task1() otherwise is very simple. At the top of the loop, it toggles the on/off state of GPIO PC13. Next, a delay is executed for 500 ms. The vTaskDelay() function requires the number of ticks to delay. It is often more convenient to specify milliseconds instead. The macro pdMS_TO_TICKS() converts milliseconds to ticks according to your FreeRTOS configuration.

This task, of course, assumes that all of the necessary setup has been done beforehand. This is taken care of by the main program, illustrated in Listing 5-4.

Listing 5-4. blinky2/main.c, main() Function

```
0034: int
0035: main(void) {
0036:
0037:    rcc_clock_setup_in_hse_8mhz_out_72mhz(); // For "blue pill"
0038:
0039:    rcc_periph_clock_enable(RCC_GPIOC);
0040:    gpio_set_mode(
0041:        GPIOC,
```

```
0042:         GPIO_MODE_OUTPUT_2_MHZ,
0043:         GPIO_CNF_OUTPUT_PUSHPULL,
0044:         GPIO13);
0045:
0046:   xTaskCreate(task1,"LED",100,NULL,configMAX_PRIORITIES-1,NULL);
0047:   vTaskStartScheduler();
0048:
0049:   for (;;);
0050:   return 0;
0051: }
```

The `main()` program is defined as returning an `int` in lines 34 and 35, even though the main program should never return in this MCU context. This simply satisfies the compiler that it is conforming to POSIX (Portable Operating System Inferface) standards. The return statement in line 50 is never executed.

Line 37 illustrates something new—the establishment of the CPU clock speed. For your Blue Pill device, you'll normally want to invoke this function for best performance. It configures clocks so that the HSE (high-speed external oscillator) is using an 8 MHz crystal, multiplied by 9 (implied) by a PLL (phase-locked loop), to arrive at a CPU clock rate of 72 MHz. Without this call, we would rely on the RC clock (resistor/capacitor clock).

Line 39 enables the GPIO clock for port C. This is the first step in the ducks-in-a-row setup for GPIO `PC13`, which drives the built-in LED. Lines 40–44 define the remaining ducks in a row so that `PC13` is an output pin, at 2 MHz, in push/pull configuration.

Line 46 creates a new task, using our function named `task1()`. We give the task a symbolic name of "LED," which can be a name of your choosing. The third argument specifies how many stack *words* are required for the stack space. Notice the emphasis on "words." For the STM32 platform, a word is four bytes. Estimating stack space is often tricky, and there are ways to measure it (see Chapter 21, "Troubleshooting"). For now, accept that 400 bytes (100 words) is enough.

The fourth argument in line 46 points to any data that you want to pass to your task. We don't need to here, so we specify `NULL`. This pointer is passed to the argument `args` in `task1()`. The fifth argument specifies the task priority. We only have one task in this example (aside from the main task). We simply give it a high priority. The last argument allows a task handle to be returned if we provide a pointer. We don't need the handle returned, so `NULL` is supplied.

Creating a task alone is not enough to start it running. You can create several tasks before you start the task scheduler. Once you invoke the FreeRTOS function vTaskStartScheduler(), the tasks will start from the function address that you named in argument one.

Exercise some care in choosing functions to call prior to the start of the task scheduler. Some of the more advanced functions may only be called after the scheduler is running. There are still others that can only be called prior to the scheduler being started. Check the FreeRTOS documentation when necessary.

Once the task scheduler is running, it never returns from line 47 of Listing 5-4 unless the scheduler is stopped. In case it does return, it is customary to put a forever loop (line 49) after the call to prevent it from returning from main (line 50).

Build and Test blinky2

With your programmer hooked up to your device, perform the following:

```
$ make clobber
$ make
# make flash
```

The make clobber deletes any built or partially built components so that a plain make will completely recompile in the project again. The make flash will invoke the st-flash utility to write the new program to your device. Press the Reset button if necessary, but it may start on its own.

The code shows that the built-in LED should change state every 500 ms. If you have a scope, you can confirm that this does indeed happen (scope pin PC13). This not only confirms that the program works as intended, but also confirms that our FreeRTOS.h file has been properly configured.

Execution

The example program is rather simple, but let's summarize the high-level activities of what is happening:

- The function task1() is concurrently executing, toggling the built-in LED on and off. This is timed by the timer facility through the use of vTaskDelay().

- The `main()` function has called `vTaskStartScheduler()`. This gives control to the FreeRTOS scheduler, which starts and switches various tasks. The main thread will continue to execute within FreeRTOS (within the scheduler) unless a task stops the scheduler.

Task `task1()` has a stack allocated from the heap to execute with (we gave it 100 words). If that task were ever deleted, this storage would be returned to the heap. The main task is currently executing within the FreeRTOS scheduler, using the stack it was given.

While these may seem like elementary points to make, it is important to know where the resources are allocated. Larger applications need to carefully allocate memory and CPU so that no task becomes starved. This also outlines the overall control structure that is operating.

The use of preemptive multi-tasking requires new responsibility. Sharing data between tasks requires thread-safe disciplines to be used. This simple example skirts the issue because there is only one task. Later projects will require inter-task communication.

FreeRTOSConfig.h

Each of the projects found in the `~/stm32f103c8t6/rtos` subdirectories has its own copy of `FreeRTOSConfig.h`. This is by design since this configures your RTOS resources and features, which may vary by project. This permits some projects to leave out FreeRTOS features that they don't require, resulting in a smaller executable. In other cases, there can be differences in timing, memory allocation, and other RTOS-related features.

Listing 5-1, line 5, illustrated that `FreeRTOS.h` is included. This file in turn causes your local `FreeRTOSConfig.h` file to be included. Let's now examine some of the important configuration elements within the `FreeRTOSConfig.h` file, shown in Listing 5-5.

Listing 5-5. Some Configuration Macros Defined in the FreeRTOSConfig.h File

```
0088: #define configUSE_PREEMPTION      1
0089: #define configUSE_IDLE_HOOK       0
0090: #define configUSE_TICK_HOOK       0
0091: #define configCPU_CLOCK_HZ        ( ( unsigned long ) 72000000 )
0092: #define configSYSTICK_CLOCK_HZ    ( configCPU_CLOCK_HZ / 8 )
```

```
0093: #define configTICK_RATE_HZ         ( ( TickType_t ) 250 )
0094: #define configMAX_PRIORITIES       ( 5 )
0095: #define configMINIMAL_STACK_SIZE   ( ( unsigned short ) 128 )
0096: #define configTOTAL_HEAP_SIZE      ( ( size_t ) ( 17 * 1024 ) )
0097: #define configMAX_TASK_NAME_LEN    ( 16 )
0098: #define configUSE_TRACE_FACILITY   0
0099: #define configUSE_16_BIT_TICKS     0
0100: #define configIDLE_SHOULD_YIELD    1
0101: #define configUSE_MUTEXES          0
0102: #define configCHECK_FOR_STACK_OVERFLOW 1
```

The most important of these configuration macros is perhaps the configUSE_ PREEMPTION macro. When set to non-zero, it indicates that we want preemptive scheduling in FreeRTOS. There are two hook functions, which were not used, so configUSE_IDLE_HOOK and configUSE_TICK_HOOK are set to zero.

The following three macros configure FreeRTOS so that it can compute the correct timing for us:

```
0091: #define configCPU_CLOCK_HZ         ( ( unsigned long ) 72000000 )
0092: #define configSYSTICK_CLOCK_HZ   ( configCPU_CLOCK_HZ / 8 )
0093: #define configTICK_RATE_HZ         ( ( TickType_t ) 250 )
```

These declarations indicate a 72 MHz CPU clock rate, a system timer counter that will increment every 8 CPU cycles, and that we want a system click interrupt to happen 250 times per second (every 4 ms). If you get these values incorrect then FreeRTOS won't get timings or delays correct.

The value of configMAX_PRIORITIES defines the maximum number of priorities that will be supported. Each priority level requires RAM within RTOS, so the levels should not be set higher than necessary.

The minimum stack size (in *words*) specifies how much space the FreeRTOS idle task needs. This should not normally be modified. The heap size in bytes declares how much RAM can be dynamically allocated. In this example, the 17K of SRAM out of the 20K total is available as heap:

```
0095: #define configMINIMAL_STACK_SIZE   ( ( unsigned short ) 128 )
0096: #define configTOTAL_HEAP_SIZE      ( ( size_t ) ( 17 * 1024 ) )
```

The `configIDLE_SHOULD_YIELD` macro should be enabled if you want the idle task to invoke another task that is ready to run. Finally, `configCHECK_FOR_STACK_OVERFLOW` was enabled for this application so that we could demonstrate the function `vApplicationStackOverflowHook()` in Listing 5-2. If you don't need this functionality, turn it off by setting it to zero.

The following macros are examples of other customizations. In our example program, we never use the `vTaskDelete()` function, for example, so `INCLUDE_vTaskDelete` is set to zero. This reduces the overhead of the compiled FreeRTOS code. We do, however, need the `vTaskDelay()` function, so the macro `INCLUDE_vTaskDelay` is configured as 1:

```
0111: #define INCLUDE_vTaskPrioritySet        0
0112: #define INCLUDE_uxTaskPriorityGet       0
0113: #define INCLUDE_vTaskDelete             0
0114: #define INCLUDE_vTaskCleanUpResources   0
0115: #define INCLUDE_vTaskSuspend            0
0116: #define INCLUDE_vTaskDelayUntil         0
0117: #define INCLUDE_vTaskDelay              1
```

FreeRTOS Naming Convention

The FreeRTOS naming convention differs from that used by libopencm3. The FreeRTOS group uses a unique naming convention for variables, macros, and functions. As a software developer myself, I don't recommend the practice of including type information in named entities. The problem is that types can change as the project matures or is ported to a new platform. When that happens, you're faced with two ugly choices, as follows:

1. Leave the entity names as they are and live with the fact that the type information is not correct.

2. Edit all of the name references to reflect the new type.

The UNIX convention of including a "p" to indicate a pointer variable is usually acceptable because it is uncommon for a variable to change from a pointer to an instance. Yet this too can happen in C++, where reference variables can be used.

Despite this odd naming convention, let's not waste time rewriting FreeRTOS or putting layers around it. Here, I'll simply identify the conventions that they have used so that it is easier for you to make use of their API in Table 5-1.

Table 5-1. *FreeRTOS Type Prefix Characters*

Prefix	Description
v	void (function return value)
c	char type
s	short type
l	long type
x	BaseType_t and any other type not covered
u	unsigned type
p	pointer

So, if the variable has a type of unsigned char, they will use the prefix "uc." If the variable is a pointer to an unsigned character, they will use "puc."

You have already seen the function named vTaskDelay(), which indicates that there is no return value (void). The FreeRTOS function named xQueueReceive() returns a type BaseType_t, which is why the function name prefix is "x."

FreeRTOS Macros

FreeRTOS writes macro names with a prefix to indicate where they are defined. Table 5-2 lists these.

Table 5-2. *Macro Prefixes Used by FreeRTOS*

PREFIX	Example	Source
port	portMAX_DELAY	portable.h
task	taskENTER_CRITICAL()	task.h
pd	pdTRUE	projdefs.h
config	configUSE_PREEMPTION	FreeRTOSConfig.h
err	errQUEUE_FULL	projdefs.h

Summary

The blink program, as simple as it is, was presented running under FreeRTOS as a task. And yet it still remained uncomplicated and provided a reliable timing, changing state every 500 ms.

We also saw how to configure the CPU clock rate so that we would not have to accept a default RC clock for it. This is important for FreeRTOS so that its timing will be accurate. An optional hook function for capturing a stack overrun event was illustrated. FreeRTOS configuration and conventions were covered, and you saw how easy it is to create preemptive tasks.

EXERCISES

1. How many tasks are running in blinky2?

2. How many threads of control are operating in blinky2?

3. What would happen to the blink rate of blinky2 if the value of configCPU_CLOCK_HZ were configured as 36000000?

4. Where does task1's stack come from?

5. Exactly when does task1() begin?

6. Why is a message queue needed?

Change to the project in `stm32/rtos/blinky`, build it, and run it. Then, answer the following:

7. Even though it uses an execution delay loop, why does it seem to work with a nearly 50 percent duty cycle?

8. How difficult is it to estimate how long the LED on PC13 is on for? Why?

9. Using a scope, measure the on and off times of PC13 (or count how many blinks per second and compute the inverse). How many milliseconds is the LED on for?

10. If another task were added to this project that consumed most of the CPU, how would the blink rate be affected?

11. Add to the file `main.c` a task2 that does nothing but execute `__asm__` (`"nop"`) in a loop. Create that task in `main()` prior to starting the scheduler. How did that impact the blink rate? Why?

CHAPTER 6

USART

The Blue Pill PCB provides one GPIO-controlled LED to communicate by. Needless to say, this would be limiting if it were all you had. Perhaps the best early-development peripheral to pursue for communication is the USART (Universal Synchronous/Asynchronous Receiver/Transmitter).

This chapter will examine how to coax a STM32 USART to speak to your desktop through a USB serial adapter cable. A second project will demonstrate the same USART using two FreeRTOS tasks and a message queue.

USART/UART Peripheral

Within this book and technical literature at large, you will see the terms USART and UART used almost interchangeably. The difference between the two is in capability: USART is short for Universal *Synchronous*/Asynchronous Receiver/Transmitter. The UART moniker drops the synchronous function from the designation.

USART/UART peripherals send data serially over a wire. One wire is used for sending (TX) and another for receiving (RX) data. There is implied a common-ground connection between the two endpoints. Synchronous communication sometimes requires one end to act as the master and provide a clock signal. Asynchronous communication does not use a separate clock signal but does require both ends to agree precisely on a clock rate—known as the baud rate. Asynchronous communication begins with a start bit and ends with a stop bit for each character.

The USART peripherals provided by the STM32F103 are quite flexible. These can indeed function as USART or UART, depending upon configuration. This chapter will focus on the asynchronous mode for simplicity, and thus the name UART applies to the remainder of this chapter.

Asynchronous Data

Figure 6-1 provides an annotated scope trace of an asynchronous byte 0x65 being transmitted. This TTL (Transistor Transistor Logic) signal starts at the left with the line idle high (near 5 volts). The beginning of the character is marked by a low bit (near zero volts), known as the *start bit*. This alerts the receiver that data bits are following, with the least significant bits first (little endian). This example was an 8-bit value. The end of the character is marked by a *stop bit*. If the stop bit is not seen by the receiver, then an error is flagged.
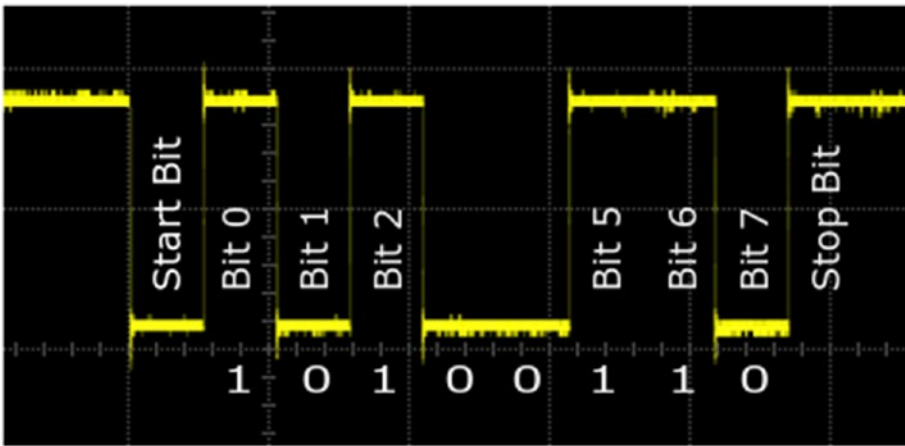
Figure 6-1. *Annotated scope trace of the UART signal for the value 0x65. Note how the least significant bits are sent first.*

Values being sent can be configured to be 8 or 9 bits in length. The last bit is the parity bit when enabled. The stop bit(s) end the transmission of a character and are configured as 0.5, 1, 1.5, or 2 bits in length.

USB Serial Adapters

A USB TTL serial adapter is an extremely helpful thing to own when working with microcontrollers. With very little hookup, you can use a terminal program on your desktop to communicate with your STM32. This eliminates the need for a costly LCD screen and keyboard.

If you haven't acquired one yet, here are some guidelines for what to look for:

- It must be a "TTL" adapter (signals at +5 volts or +3.3 volts).

- The USB device is supported by your operating system.

- The unit supports hardware flow control (RTS and CTS).

The TTL distinction is important. Normal RS-232 adapters operate at plus and minus about 3 volts or more. These *cannot* be wired directly to your STM32.

The *TTL* serial adapters, on the other hand, signal between zero and +5 volts and can be used with any of the *5-volt-tolerant inputs*. Fortunately, ST Microelectronics arranged that the receive line (RX) for UART 1 and 3 has 5-volt-tolerant inputs. Sending from the 3.3-volt STM32 works fine because the high signal is well above the threshold needed to be received as a 1-bit by the adapter.

Figure 6-2 illustrates one that is used by the author. These can be purchased for around $3 US on eBay. Be sure to get a unit that supports hardware flow control. These will include connections for RTS and CTS. Without hardware flow control, you won't be able to support higher rates like 115,200 baud without losing data. Be careful about FTDI units. In the past there have been reports of FTDI (FTDI Chip) drivers bricking FTDI clones. It is best to get a genuine FTDI unit or to avoid units claiming FTDI compatibility.
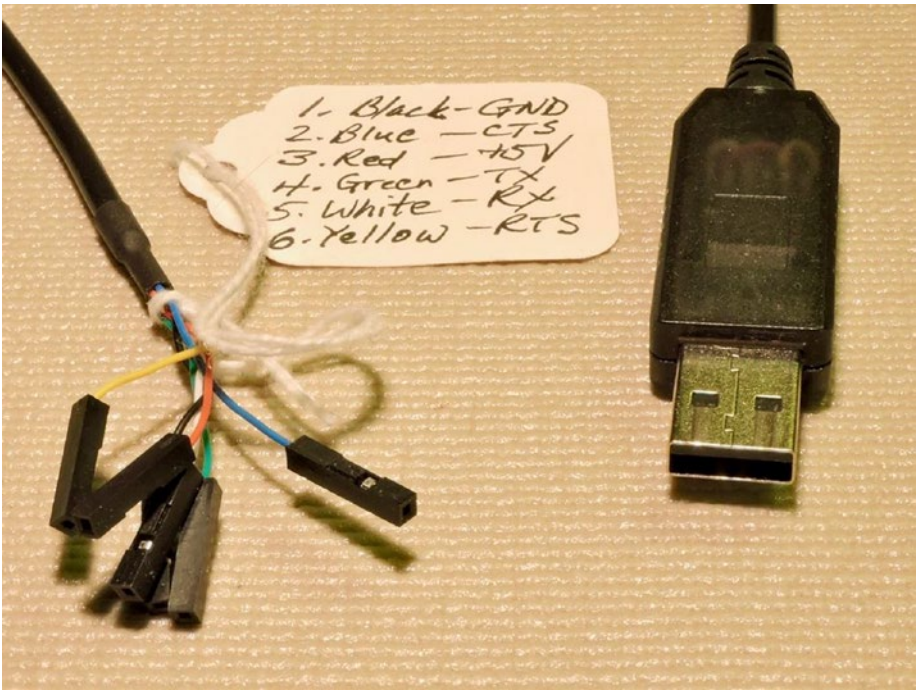
Figure 6-2. *Example USB TTL serial adapter cable. A tag was added as a colored-wires legend.*

Hookup

The two projects featured in this chapter require you to attach a USB TTL serial adapter so that your desktop can view the output. Figure 6-3 shows the hookup required.
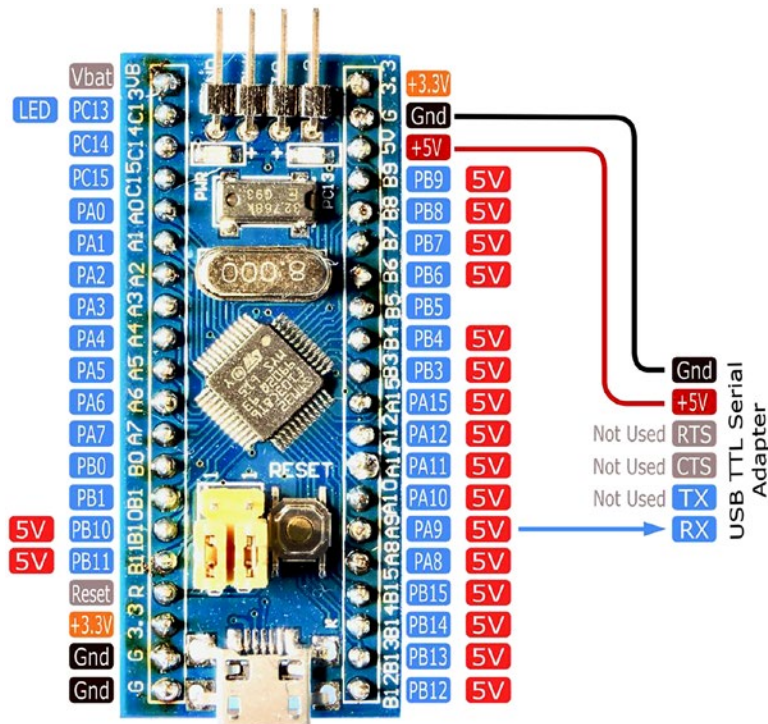
Figure 6-3. *USB TTL serial hookup for outgoing communication. No flow control is used in this example.*

The baud rate used in this chapter is 38,400, which is a relatively low speed. This allows us to avoid flow control for simplicity for our first demo. You should be able to power your device from the serial adapter's +5 volt line, as shown. Connect that supply to the +5 volt input on your Blue Pill so that the onboard regulator will provide the MCU 3.3 volts.

If this is not possible, then power the device separately. Be sure to make a common ground connection between the power source, the MCU, and the serial adapter.

Finally, note that only one data connection is required for these particular demos. This is because these demonstration programs only transmit and do not receive data from the desktop.

Project uart

Change to the following source directory:

```
$ cd ~/stm32f103c8t6/rtos/uart
```

Make sure that the USB serial adapter has been unplugged before attaching the programmer. With the programmer ready, build and flash it as follows:

```
$ make clobber
$ make
$ make flash
```

Once flashed and the device starts to run the program, it will output text. To see it, you will need to use a terminal emulator on your desktop. Disconnect the ST-Link V2 programmer and attach the USB serial adapter. Figure 6-2 illustrates the connections. With the power applied from the serial adapter, you should see the STM32 device's power LED light and the PC13 LED flashing.

I will be using the older minicom terminal program in this text, but another good program is putty. Use your system's package manager to install either of these if necessary.

To use your serial adapter, you will need to know the operating system–specific device pathname or COM port. For Mac or Linux, you might be able to discover it just by looking into the /dev directory. On the Mac, the device will show up with a /dev/cu prefix when it is plugged in and active (otherwise, look for /dev/ttyusbserial*). When you unplug it, this device name will disappear.

Using minicom, you'll need to configure the communication port first by supplying the -s option:

```
$ minicom -s

    +-----[configuration]------+
    | Filenames and paths      |
    | File transfer protocols  |
    |>Serial port setup        |
    | Modem and dialing        |
    | Screen and keyboard      |
    | Save setup as dfl        |
    | Save setup as..          |
    | Exit                     |
    | Exit from Minicom        |
    +--------------------------+
```

Scroll down to "Serial port setup" and press Enter:

```
+----------------------------------------------------------------+
| A -     Serial Device      : /dev/cu.usbserial-A703CYQ5        |
| B - Lockfile Location      : /usr/local/Cellar/minicom/2.7/var |
| C -    Callin Program      :                                   |
| D -  Callout Program       :                                   |
| E -     Bps/Par/Bits       : 2400 8O1                          |
| F - Hardware Flow Control : No                                 |
| G - Software Flow Control : No                                 |
|                                                                |
|    Change which setting?                                       |
+----------------------------------------------------------------+
```

Type in "A" if the device pathname shown is incorrect. Make sure that flow control is disabled by typing "F" and/or "G" if necessary. Finally, type "E" to change the port settings:

```
+----------------+---------[Comm Parameters]----------+-------+
| A -     Serial De|                                  |       |
| B - Lockfile Loc|     Current: 38400 8N1            |2.7/var|
| C -    Callin Pro| Speed            Parity     Data |       |
| D -  Callout Pro| A: <next>        L: None    S: 5 |       |
| E -     Bps/Par/B| B: <prev>        M: Even    T: 6 |       |
| F - Hardware Flo| C:   9600        N: Odd     U: 7 |       |
| G - Software Flo| D:  38400        O: Mark    V: 8 |       |
|                 | E: 115200        P: Space        |       |
|    Change which |                                  |       |
+----------------| Stopbits                         |-------+
        | Screen a| W: 1             Q: 8-N-1        |
        | Save set| X: 2             R: 7-E-1        |
        | Save set|                                  |
        | Exit    |                                  |
        | Exit fro| Choice, or <Enter> to exit?      |
        +---------+----------------------------------+
```

Once the "Comm Parameters" panel is shown, you can type "Q" to choose "8-N-1" and then type "D" to set the baud rate to 38,400. Then, press Enter twice to return to the main menu.

At the main menu, choose "Save setup as…" to save your settings for next time. Let's use "chap6" for the name and press Enter. If you have difficulty saving, it is likely because the packaged minicom has set the directory to a location that you lack permissions on (a big sigh from the author!). In this case, you should use the full pathname, starting with slash, to override the directory component.

```
+----------------------------------------+
|Give name to save this configuration?   |
|> chap6                                 |
+----------------------------------------+
```

If your device started after being flashed, you might see an incomplete first line, but the remainder of the output should be similar to the following:

```
Welcome to minicom 2.7

OPTIONS:
Compiled on Sep 17 2016, 05:53:15.
Port /dev/cu.usbserial-A703CYQ5, 16:30:48

Press Meta-Z for help on special keys

UVWXYZ
0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789:;<=>?@
```

The program is designed to slowly write repeating lines, with time in between each character. Being slow like this avoids the need for flow control.

If you need to restart minicom, you can now use your saved settings as follows:

```
$ minicom chap6
```

On the Mac, minicom can develop USB driver problems if you just unplug the adapter without first exiting the program. To exit minicom, use ESC-X (on some systems you must use Control-A-X instead).

```
+----------------------+
|    Leave Minicom?    |
|     Yes        No    |
+----------------------+
```

Yes should be highlighted by default, allowing you to just press Enter. If you don't see this, then you need to try again. Press X immediately after pressing ESC (or Control-A), since this operation is time sensitive. Once minicom has closed the USB driver and exited, it is safe to unplug the serial adapter. If you spoil a USB port, you can either use another port or reboot.

Project

Listing 6-1 illustrates the main program uart.c. The only thing new in the main function is the call to a separate setup routine named uart_setup() in line 94.

Listing 6-1. Listing of ~/stm32f103c8t6/rtos/uart/uart.c Main Program

```
0081: int
0082: main(void) {
0083:
0084:    rcc_clock_setup_in_hse_8mhz_out_72mhz(); // Blue pill
0085:
0086:    // PC13:
0087:    rcc_periph_clock_enable(RCC_GPIOC);
0088:    gpio_set_mode(
0089:        GPIOC,
0090:                    GPIO_MODE_OUTPUT_2_MHZ,
0091:                    GPIO_CNF_OUTPUT_PUSHPULL,
0092:                    GPIO13);
0093:
0094:    uart_setup();
0095:
```

```
0096:    xTaskCreate(task1,"task1",100,NULL,configMAX_PRIORITIES-1,NULL);
0097:    vTaskStartScheduler();
0098:
0099:    for (;;);
0100:    return 0;
0101: }
```

Listing 6-2 illustrates this setup code for UART1. Notice that two clock systems are enabled in lines 31 and 32. The TX output of UART1 comes out to PA9 by default (we'll examine alternate-function I/O later in the book), so the GPIOA subsystem needs its clock enabled. The USART1 peripheral also needs a clock, which is enabled in line 32.

Lines 35 to 38 use function `gpio_set_mode()` to configure that output pin. Note that the higher-rate `GPIO_MODE_OUTPUT_50_MHZ` option is chosen here to allow sharper signal changes. Note especially that the macro `GPIO_CNF_OUTPUT_ALTFN_PUSHPULL` specifies that it is *non-GPIO* (the ALTFN part) and that the output should use a push/pull configuration. The ALTFN aspect is super critical here—a common mistake is to choose the GPIO form (apologies for harping on it).

Line 38 specifies libopencm3 macro `GPIO_USART1_TX`, which on the STM32F103 platform equates to pin PA9. Using the macro `GPIO13` would have been equally valid, although the code is more portable as given.

Line 40 calls upon `usart_set_baudrate()` to establish the baud rate of 38,400. This function calculates a divisor necessary to arrive at the approximate value for the baud rate. Odd-valued baud rates may lack the accuracy that standard baud rates enjoy.

Line 41 uses function `usart_set_databits()` to configure how many bits each character will contain. Here, the valid choices are 8 or 9. With parity enabled, this implies 7 or 8 bits of data, respectively.

One stop bit is configured in line 42, and the peripheral is set for transmit-only in line 43. Line 44 indicates no parity bit will be sent, and line 45 indicates that no hardware flow control will be used. Finally, line 46 enables the peripheral for operation.

Listing 6-2. Listing of uart_setup( ) in stm32/rtos/uart/uart.c

```
0028: static void
0029: uart_setup(void) {
0030:
0031:    rcc_periph_clock_enable(RCC_GPIOA);
0032:    rcc_periph_clock_enable(RCC_USART1);
```

```
0033:
0034:    // UART TX on PA9 (GPIO_USART1_TX)
0035:    gpio_set_mode(GPIOA,
0036:        GPIO_MODE_OUTPUT_50_MHZ,
0037:        GPIO_CNF_OUTPUT_ALTFN_PUSHPULL,
0038:        GPIO_USART1_TX);
0039:
0040:    usart_set_baudrate(USART1,38400);
0041:    usart_set_databits(USART1,8);
0042:    usart_set_stopbits(USART1,USART_STOPBITS_1);
0043:    usart_set_mode(USART1,USART_MODE_TX);
0044:    usart_set_parity(USART1,USART_PARITY_NONE);
0045:    usart_set_flow_control(USART1,USART_FLOWCONTROL_NONE);
0046:    usart_enable(USART1);
0047: }
```

As you can see, there are several UART details that require configuration, and all of these must match what you are using in the receiving desktop terminal program.

Our application task1() function sends data to another routine, uart_putc(), which is provided. Function task1() is illustrated in Listing 6-3. The task is designed to put out lines of text of the following form:

```
0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

As part of the loop, the built-in LED PC13 is toggled in line 65. This gives us confidence that the program is running, should there be trouble in getting the UART output to the desktop. In between each character, the task waits 200 ms to slow the sending down (line 66). This saves us from having to deal with flow control for now. Lines 67 to 74 transmit the character (line 67 increments c) by calling uart_putc().

Listing 6-3. The Function task1( ) of the Application Program

```
0060: static void
0061: task1(void *args __attribute__((unused))) {
0062:   int c = '0' - 1;
0063:
```

```
0064:    for (;;) {
0065:        gpio_toggle(GPIOC,GPIO13);
0066:        vTaskDelay(pdMS_TO_TICKS(200));
0067:        if ( ++c >= 'Z' ) {
0068:            uart_putc(c);
0069:            uart_putc('\r');
0070:            uart_putc('\n');
0071:            c = '0' - 1;
0072:        } else  {
0073:            uart_putc(c);
0074:        }
0075:    }
0076: }
```

Listing 6-4 illustrates function uart_putc(), which simply calls upon the libopencm3 routine usart_send_blocking(). As implied by the function name, control does not return until the USART is ready to accept more data. In the next example, a more task-friendly approach will be applied.

Listing 6-4. The uart_putc( ) Function uart.c

```
0052: static inline void
0053: uart_putc(char ch) {
0054:   usart_send_blocking(USART1,ch);
0055: }
```

Essentially, this example boils down to the following main points:

1. How to configure and enable the UART for transmission

2. How to apply libopencm3 to send data to UART1

Apart from the fact that the main thread is running the task scheduling and the application is running in function task1(), the design is still inelegant. The example works as presented, but let's partition it a little more and correct the deficiencies in design.

Project uart2

Change now to the following project directory:

```
$ cd ~/stm32f103c8t6/rtos/uart2
```

The source module uart.c in this project has some enhancements in it. First, there is a new include file named queue.h (line 15), which is provided by FreeRTOS. This allows a message queue handle uart_txq to be declared at line 21 (Listing 6-5).

Listing 6-5. Include Files Used by stm32/rtos/uart2/uart.c

```
0013: #include <FreeRTOS.h>
0014: #include <task.h>
0015: #include <queue.h>
0016:
0017: #include <libopencm3/stm32/rcc.h>
0018: #include <libopencm3/stm32/gpio.h>
0019: #include <libopencm3/stm32/usart.h>
0020:
0021: static QueueHandle_t uart_txq;    // TX queue for UART
```

The setup routine remains the same except for the creation of the message queue at line 47 of Listing 6-6. The call creates a message queue that will contain a maximum of 256 messages, each with a message length of one byte. The variable uart_txq then receives a valid handle.

Listing 6-6. The uart_setup( ) Function

```
0026: static void
0027: uart_setup(void) {
0028:
0029:   rcc_periph_clock_enable(RCC_GPIOA);
0030:   rcc_periph_clock_enable(RCC_USART1);
0031:
0032:   // UART TX on PA9 (GPIO_USART1_TX)
0033:   gpio_set_mode(GPIOA,
0034:       GPIO_MODE_OUTPUT_50_MHZ,
```

```
0035:          GPIO_CNF_OUTPUT_ALTFN_PUSHPULL,
0036:          GPIO_USART1_TX);
0037:
0038:   usart_set_baudrate(USART1,38400);
0039:   usart_set_databits(USART1,8);
0040:   usart_set_stopbits(USART1,USART_STOPBITS_1);
0041:   usart_set_mode(USART1,USART_MODE_TX);
0042:   usart_set_parity(USART1,USART_PARITY_NONE);
0043:   usart_set_flow_control(USART1,USART_FLOWCONTROL_NONE);
0044:   usart_enable(USART1);
0045:
0046:   // Create a queue for data to transmit from UART
0047:   uart_txq = xQueueCreate(256,sizeof(char));
0048: }
```

The routine for writing out characters is now run from function uart_task(), which is scheduled as a task in Listing 6-7.

Listing 6-7. The uart_task( ) Task Function

```
0053: static void
0054: uart_task(void *args __attribute__((unused))) {
0055:   char ch;
0056:
0057:   for (;;) {
0058:       // Receive char to be TX
0059:       if ( xQueueReceive(uart_txq,&ch,500) == pdPASS ) {
0060:                 while ( !usart_get_flag(USART1,USART_SR_TXE) )
0061:               taskYIELD();    // Yield until ready
0062:           usart_send(USART1,ch);
0063:       }
0064:       // Toggle LED to show signs of life
0065:       gpio_toggle(GPIOC,GPIO13);
0066:   }
0067: }
```

The uart_task() function operates within a loop starting at line 57. The FreeRTOS function xQueueReceive() is called to obtain a message. The last argument 500 indicates that this call should timeout after 500 ticks. By timing out, the built-in LED on PC13 can be toggled to indicate that the program is still alive. The function xQueueReceive() returns pdFAIL when it times out.

When xQueueReceive() returns pdPASS, however, the task has received a message. The message is received as a single character into variable ch in our demo. Once we have received a character from the queue, we need to send it to the UART.

Notice the while loop in lines 60 and 61. This calls FreeRTOS function taskYIELD() until the UART is able to accept another character (in line 62). Library libopencm3 provides the function usart_get_flag() to allow the testing of various status-register flags. In this manner, the status-register bit TXE (transmit empty) is tested. As long as this register indicates "not empty," we direct the scheduler to run another task by calling taskYIELD().

If we did not yield control, the function usart_send_blocking() would simply spin, waiting for the UART to become ready. If the UART didn't become ready in time, this spinning would burn up CPU time until that task's time slice ran out. This spinning would still appear to function OK for the application but would waste CPU time that might be more profitably used elsewhere. Because the TXE flag indicates that the UART is ready at line 62, we can use the usart_send() function instead.

The uart_task() and the demo_task() run concurrently. Listing 6-8 illustrates the new demo_task(), which queues up pairs of lines to be sent. Listing 6-10 illustrates the changes to the main program made to establish these two tasks.

Listing 6-8. The demo_task( ), Which Produces a Repeating Pair of Lines

```
0081: /*****************************************************
0082:  * Demo Task:
0083:  *      Simply queues up two line messages to be TX, one second
0084:  *      apart.
0085:  *****************************************************/
0086: static void
0087: demo_task(void *args __attribute__((unused))) {
0088:
```

```
0089:    for (;;) {
0090:        uart_puts("Now this is a message..\n\r");
0091:        uart_puts("  sent via FreeRTOS queues.\n\n\r");
0092:        vTaskDelay(pdMS_TO_TICKS(1000));
0093:    }
0094: }
```

The demo_task() invokes a new function, illustrated in Listing 6-9. Function uart_puts() simply calls upon uart_putc() to put each character within a string to the UART. One important point to notice is that the xQueueSend() call in line 77 will block if the queue becomes full. The third argument specifies portMAX_DELAY so that it will block forever until it succeeds. Since this is a FreeRTOS call, the function knows to yield control to another task when the queue is full.

Listing 6-9. Function uart_puts( ) Uses uart_putc() to Transmit a String of Characters to the UART

```
0072: static void
0073: uart_puts(const char *s) {
0074:
0075:    for ( ; *s; ++s ) {
0076:        // blocks when queue is full
0077:        xQueueSend(uart_txq,s,portMAX_DELAY);
0078:    }
0079: }
```

The main program in Listing 6-10 simply calls xTaskCreate() twice to establish two tasks. One executes function uart_task() in line 114, while the other executes demo_task() in line 115. The create order is unimportant, since the FreeRTOS scheduler is not started until line 117.

Listing 6-10. The Main Program for stm32/rtos/uart2/uart.c

```
0099: int
0100: main(void) {
0101:
0102:    rcc_clock_setup_in_hse_8mhz_out_72mhz(); // CPU clock is 72 MHz
0103:
```

```
0104:    // GPIO PC13:
0105:    rcc_periph_clock_enable(RCC_GPIOC);
0106:    gpio_set_mode(
0107:        GPIOC,
0108:        GPIO_MODE_OUTPUT_2_MHZ,
0109:        GPIO_CNF_OUTPUT_PUSHPULL,
0110:        GPIO13);
0111:
0112:    uart_setup();
0113:
0114:        xTaskCreate(uart_task,"UART",100,NULL,configMAX_PRIORITIES-
1,NULL);
0115:        xTaskCreate(demo_task,"DEMO",100,NULL,configMAX_PRIORITIES-
2,NULL);
0116:
0117:    vTaskStartScheduler();
0118:    for (;;);
0119:    return 0;
0120: }
```

This is a general summary of the operation of the program:

1. Task demo_task() calls upon a routine uart_puts() to send
    strings of text to the UART, one second apart.

2. The function uart_puts() invokes uart_putc() to queue the
    characters in a message queue referenced by handle uart_txq. If
    the queue is full, control of demo_task() yields.

3. Task uart_task() unqueues characters received from the queue
    referenced by handle uart_txq.

4. Each character received is delivered to the UART to be sent,
    provided that it is ready. When the UART is busy, the control of the
    task yields.

While this has been a simple example, we see the elegance of FreeRTOS in action. One task produces while another consumes. The control loops for both are trivial. By partitioning an application into tasks, we break the problem into manageable components. We see that inter-task communication can be safely accomplished through a FreeRTOS message queue.

USART API

For your reference, let's list the API (Application Programming Interface) used in this chapter and document its arguments. Additionally, advanced API functions are included to keep the reference in one place.

For the functions listed, some arguments need special values, which are supplied by defined libopencm32 macros. They are listed in Tables 6-1 through 6-7. The table caption lists the argument name that the values refer to. For example, Table 6-2 lists the valid macro names for the parity argument.

Table 6-1 lists the different USARTs that are available to the STM32F103C8T6 device. The default pins are listed for each function. For example, USART2 receives on PA3 by default unless alternate-function I/O configuration has been applied.

Table 6-1. *USARTS Available to the STM32F103C8T6 (Argument usart)*

USART	Macro	5V	TX	RX	CTS	RTS
1	USART1	Yes	PA9	PA10	PA11	PA12
2	USART2	*No*	PA2	PA3	PA0	PA1
3	USART3	Yes	PB10	PB11	PB14	PB12

Table 6-2. *USART Parity Macros (Argument Parity)*

Macro	Description
USART_PARITY_NONE	No parity
USART_PARITY_EVEN	Even parity
USART_PARITY_ODD	Odd parity
USART_PARITY_MASK	Mask

Table 6-3. *USART Operation Mode Macros (Mode Argument)*

Macro	Description
USART_MODE_RX	Receive only
USART_MODE_TX	Transmit only
USART_MODE_TX_RX	Transmit and receive
USART_MODE_MASK	Mask

Table 6-4. *USART Stop Bit Macros (Argument Stopbits)*

Macro	Description
USART_STOPBITS_0_5	0.5 stop bits
USART_STOPBITS_1	1 stop bit
USART_STOPBITS_1_5	1.5 stop bits
USART_STOPBITS_2	2 stop bits

Table 6-5. *USART Flow Control Macros*

Macro	Description
USART_FLOWCONTROL_NONE	No hardware flow control
USART_FLOWCONTROL_RTS	RTS hardware flow control
USART_FLOWCONTROL_CTS	CTS hardware flow control
USART_FLOWCONTROL_RTS_CTS	RTS and CTS hardware flow control
USART_FLOWCONTROL_MASK	Mask

Table 6-6. *USART Data Bits (Bits Argument)*

Value	Data Bits (No Parity)	Data Bits (With Parity)
8	8	7
9	9	8

Table 6-7. *USART Status Flag Bit Macros (Flag Argument)*

Macro	Flag Description
USART_SR_CTS	Clear to send flag
USART_SR_LBD	LIN break-detection flag
USART_SR_TXE	Transmit data buffer empty
USART_SR_TC	Transmission complete
USART_SR_RXNE	Read data register not empty
USART_SR_IDLE	Idle line detected
USART_SR_ORE	Overrun error
USART_SR_NE	Noise error flag
USART_SR_FE	Framing error
USART_SR_PE	Parity error

Include Files

```
#include <libopencm3/stm32/rcc.h>
#include <libopencm3/stm32/usart.h>
```

Clocks

```
rcc_periph_clock_enable(RCC_GPIOx);
rcc_periph_clock_enable(RCC_USARTn);
```

Configuration

```
void usart_set_mode(uint32_t usart, uint32_t mode);
void usart_set_baudrate(uint32_t usart, uint32_t baud);
void usart_set_databits(uint32_t usart, uint32_t bits);
void usart_set_stopbits(uint32_t usart, uint32_t stopbits);
void usart_set_parity(uint32_t usart, uint32_t parity);
void usart_set_flow_control(uint32_t usart, uint32_t flowcontrol);
void usart_enable(uint32_t usart);
void usart_disable(uint32_t usart);
```

DMA

```
void usart_enable_rx_dma(uint32_t usart);
void usart_disable_rx_dma(uint32_t usart);
void usart_enable_tx_dma(uint32_t usart);
void usart_disable_tx_dma(uint32_t usart);
```

Interrupts

```
void usart_enable_rx_interrupt(uint32_t usart);
void usart_disable_rx_interrupt(uint32_t usart);
void usart_enable_tx_interrupt(uint32_t usart);
void usart_disable_tx_interrupt(uint32_t usart);
void usart_enable_error_interrupt(uint32_t usart);
void usart_disable_error_interrupt(uint32_t usart);
```

Input/Output/Status

```
bool usart_get_flag(uint32_t usart, uint32_t flag)
void usart_send(uint32_t usart, uint16_t data)
uint16_t usart_recv(uint32_t usart)
```

Ducks-in-a-Row

With the exception of interrupts and DMA, the following is a summary of the ducks that must be lined up to make your UART peripheral functional:

1. Enable the appropriate GPIO clocks for *all* involved I/O pins: `rcc_periph_clock_enable(RCC_GPIOx)`.

2. Enable the clock for your selected UART peripheral: `rcc_periph_clock_enable(RCC_USARTn)`.

3. Configure the mode of your I/O pins with `gpio_set_mode()`.

   a. For output pins, choose `GPIO_CNF_OUTPUT_ALTFN_PUSHPULL` for the third argument (note the `ALTFN`).

   b. For inputs, choose `GPIO_CNF_INPUT_PULL_UPDOWN` or `GPIO_CNF_INPUT_FLOAT`.

4. usart_set_baudrate()

5. usart_set_databits()

6. usart_set_stopbits()

7. usart_set_mode()

8. usart_set_parity()

9. usart_set_flow_control()

10. usart_enable()

FreeRTOS

In this chapter, we've made use of a few FreeRTOS API functions, some of which we have seen before. They'll be summarized here for your convenience.

Tasks

The following are *task*-related FreeRTOS functions that we have used to create tasks, start the scheduler, and delay execution, respectively:

```
BaseType_t xTaskCreate(
    TaskFunction_t pvTaskCode,    // function ptr
    const char * const pcName,    // string name
    unsigned short usStackDepth,  // stack size in words
    void *pvParameters,           // Pointer to argument
    uBaseType_t uxPriority,       // Task priority
    TaskHandle_t *pxCreatedTask   // NULL or pointer to task handle
); // Returns: pdPass or errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY

void vTaskStartScheduler(void);   // Start the task scheduler

void vTaskDelay(TickType_t xTicksToDelay);

void taskYIELD();
```

The pvTaskCode pointer value is simply a pointer to a function of the following form:

```
void my_task(void *args)
```

The value provided to args comes from pvParameters in the xTaskCreate() call. If not required, this value can be supplied with NULL. The stack depth is in *words* (4 bytes each).

Each task has an associated priority and is provided by the uxPriority argument. If you're running all tasks at the same priority, supply the value configMAX_PRIORITIES-1 or just use the value 1. Unless you need different priorities, set them all to the same value (see Chapter 21, "Troubleshooting," for reasons why). Be aware that you can create more tasks after vTaskStartScheduler() has been called, when necessary.

A commonly used macro for vTaskDelay() is the following:

```
pdMS_TO_TICKS(ms)      // Macro: convert ms to ticks
```

This converts a millisecond time into a tick count for programming convenience.

Queues

The queue API functions used in this chapter include the following:

```
QueueHandle_t xQueueCreate(
    UBaseType_t uxQueueLength,  // Max # of items
    UBaseType_t uxItemSize      // Item size (bytes)
);                              // Returns: handle else NULL

BaseType_t xQueueSend(
    QueueHandle_t xQueue,       // Queue handle
    const void *pvItemToQueue,  // pointer to item
    TickType_t xTicksToWait     // 0, ticks or portMAX_DELAY
);                              // Returns: pdPASS or errQUEUE_FULL

BaseType_t xQueueReceive(
    QueueHandle_t xQueue,       // Queue handle
    void *pvBuffer,             // Pointer to receiving item buffer
    TickType_t xTicksToWait     // 0, ticks or portMAX_DELAY
);                              // Returns: pdPASS or errQUEUE_EMPTY
```

The function xQueueCreate() allocates storage for the created queue, and its handle is returned. Argument uxQueueLength indicates the maximum number of items that can be held in the queue. The value uxItemSize specifies the size of each item.

Function xQueueSend() adds an item to the queue. The item pointed to by pvItemToQueue is copied into the queue's storage. Conversely, xQueueReceive() takes

an item from the queue and copies it to the caller's storage at the address `pvBuffer`. This buffer must be at least the size given by `uxItemSize` or memory corruption will result. If there is no item to be received, the call blocks according to `xTicksToWait`.

Summary

This chapter has demonstrated the general recipe for configuring and activating the USART in asynchronous mode. This permits the Blue Pill to send data to the desktop for debugging or any other reporting.

At the same time, a demonstration of FreeRTOS tasks and message queues was provided. This approach divided the sending and receiving sides of the application into their own separate tasks. This simplified the programming since each only needed to concern itself with its own operation. The message queue provided the conduit for inter-task communication between application tasks.

EXERCISES

1. What is the idle state of the TTL level of a USART signal?

2. USART data is provided in a big or little endian sequence?

3. What clock(s) must be enabled for UART use?

4. What does the abbreviation 8N1 stand for?

5. What happens if you provide UART data to be sent if the device is not yet empty?

6. Can tasks be created before, after, or before and after `vTaskStartScheduler()`?

7. What is the minimum buffer size determined by `xQueueReceive()`?

8. How do you specify that `xQueueSend()` should return immediately if the queue is full?

9. How do you specify that `xQueueReceive()` should block forever if the queue is empty?

10. What happens to the task if `xQueueReceive()` finds the queue empty and it must wait?

CHAPTER 7

USB Serial

One of the nice things about the STM32 MCU is the availability of the USB (Universal Serial Bus) peripheral. With USB, it is possible to communicate directly with a desktop platform in various modes. One of these flexible modes is USB's emulation of a serial link between the MCU and the desktop.

This chapter will explore the use of libopencm3 and FreeRTOS working together to provide a convenient means of communication. You will use the USB CDC class of operation (USB communication device class). This provides a very convenient means for interacting with your Blue Pill.

Blue Pill USB Issue

First, let's clear the air about the Blue Pill USB issue. What is this issue you may have read about in the Internet forums?

It turns out that the PCB is manufactured with a 10 kohm resistor (R_{10}) pullup resistor to +3.3 volts, which is incorrect. For full-speed USB, this is supposed to be 1.5 kohm. You can test this by measuring resistance with your DMM between the A12 pin on the PCB and the +3.3-volt pin. You will likely read 10 kohms.

This defect does not always prevent it from working, however. For example, I had no difficulty using USB from the STM32 to a MacBook Pro. But your mileage may vary. The hard way to correct this is to replace R_{10} on the PCB, but this is difficult because the resistor is so incredibly small.

Caution   Many people have reported in online forums that their Blue Pill USB connector has broken off or become inoperable. Exercise extra-gentle care when inserting the cable.

Correction of the issue is best accomplished by placing another resistor in *parallel* with it. Placing a 1.8 kohm resistor in parallel with the 10 kohm resistor produces a combined resistance of 1.5 kohms. Figure 7-1 illustrates how the author soldered a resistor to one of his units. The 1/8-Watt resistor is simply soldered carefully between pins A12 and the +3.3-volt pin. It's not pretty, but it works!
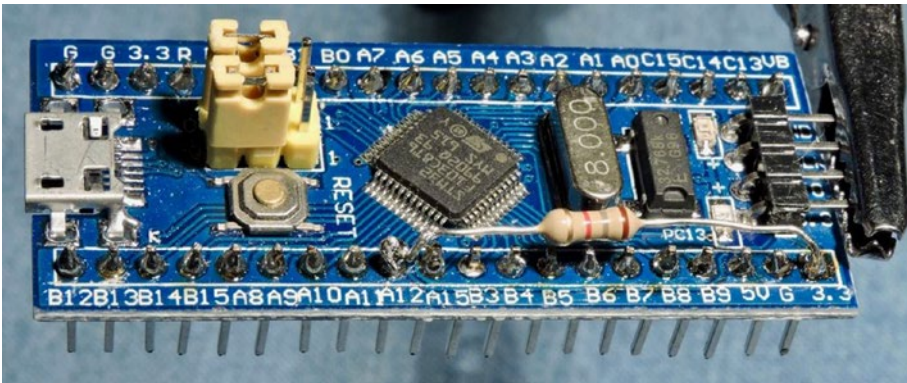
Figure 7-1. *Correcting the USB pullup by addition of a 1.8-kohm resistor*

To see how pullup resistance makes a difference, look at the scope trace in Figure 7-2. This is what the D+ line looked like with the default 10-kohm resistor.
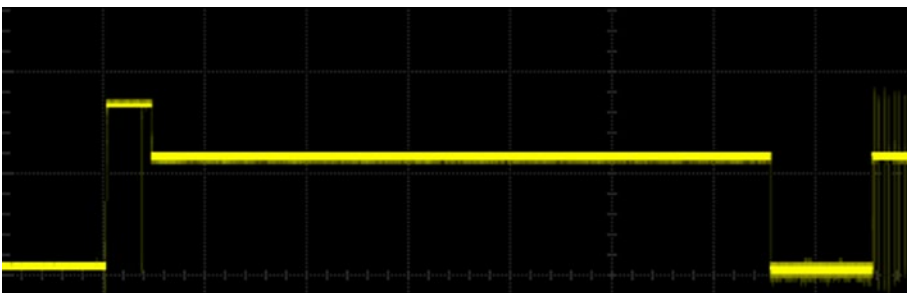
Figure 7-2. *D+ line scope trace using 10-kohm pullup resistance*

In the figure, you can see a rise at the start followed by a slump to perhaps the 70 percent level. To the right where the high-frequency signals begin, you can see that the signal rests at about the 70 percent level in between excursions. Attach this device to a different PC USB port or hub and the degradation might be worse.

Compare this to Figure 7-3, which is a scope trace after the 1.5-kohm pullup resistance was in effect.
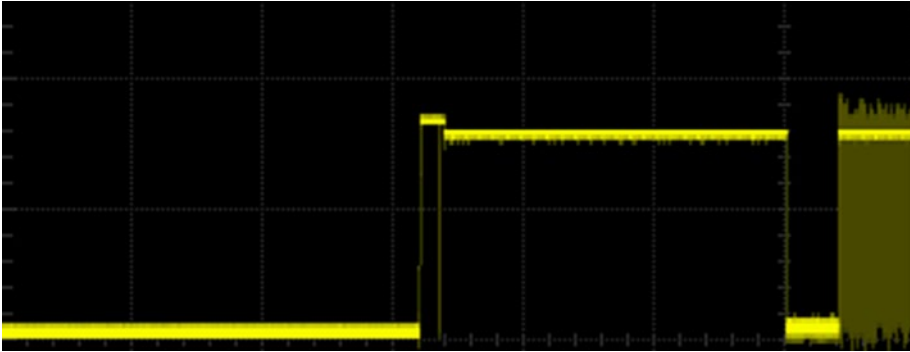
Figure 7-3. *D+ line scope trace with 1.5-kohm pullup resistance*

Ignoring capture-timing differences, you can see that the signal rests much higher, perhaps at the 90 percent level. This helps to assure improved signal thresholds.

Introduction to USB

USB is a popular means of communication from a personal computer to various peripherals, such as printers, scanners, keyboards, and a mouse. Part of its success is due to its standardization and low cost. The standard also includes USB hubs allowing the cost-effective extension of the network to accommodate additional devices.

In USB communication, the host directs all traffic. Each device is polled on a regular basis based upon its configuration and requirements. A keyboard infrequently needs to send data, for example, while a sound-recording device needs to send bulk recording data in real time. These differences are accommodated by the USB standard and are part of the device configuration.

Pipes and Endpoints

USB uses the concept of endpoints with connecting pipes to carry the data. The pipe carries the information, while the endpoints send or receive. Every USB device has at least one endpoint known as endpoint 0. This is a default and *control* endpoint, which allows host and device to configure device-specific operations and parameters. This occurs during device *enumeration*.

Figure 7-4 provides a high-level view of endpoints 0, 1, and 2 that we will be using in the example program. Technically, endpoint 0 is just one pipe. It is drawn here as two pipes because the control endpoint permits a response back to the host. All other endpoints have data travelling in one direction only. Note that the "In" and "Out" in Figure 7-4 are labeled according to the *host* controller's viewpoint.
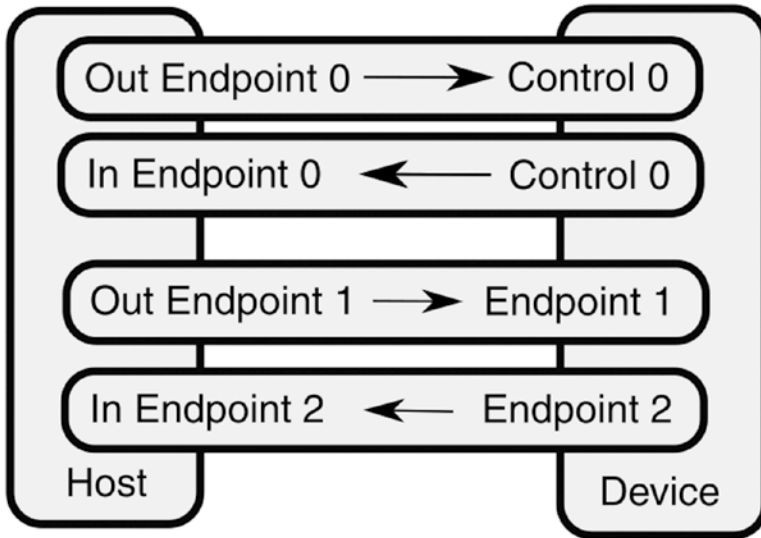
Figure 7-4. *USB pipes and endpoints*

A device may have additional endpoints, but our USB CDC example only needs two in addition to the required control endpoint 0:

- Endpoint 1 is the device's receiving endpoint (host's sending, specified as 0x01)

- Endpoint 2 is the device's sending endpoint (host's receiving, specified as 0x82)

As will be seen in the source code, bit 7 of the endpoint number indicates whether it is an input or output (with respect to the *host* controller). The value 0x82 indicates in hexadecimal that endpoint 2 (with bit 7) is sending (from the device's point of view). Unlike a TCP/IP socket, USB pipes transfer data in *one* direction only.

As you may have realized, one potentially confusing aspect of USB programming is that input and output are specified in the code from the host controller's point of view. For example, endpoint 0x82 is a receiving (input) endpoint from the host's point of view. This tends to be confusing when writing for the *device*. Be aware of that when setting up USB descriptors.

This necessarily has been a brief introduction to USB. Entire books have been written on the subject, and the interested reader is encouraged to seek them out. Our focus will be limited to the successful *use* of the USB peripheral for the benefit of our STM32. Let's get started!

USB Serial Device

With the MCU flashed and plugged into the system, you need to access it on your operating system as a serial device. This practice varies with the operating system, which complicates things slightly. The MCU source code is found in the following directory:

```
$ cd ~/stm32f103c8t6/rtos/usbcdcdemo
$ make clobber
$ make
$ make flash
```

The preceding steps will build and flash the code into your MCU device. The following sections will describe details on the desktop side of the USB conduit.

Linux USB Serial Device

Under Linux, with the STM32 flashed and plugged into a USB port, you can use the lsusb command to view the connected devices:

```
$ lsusb
Bus 002 Device 003: ID 0483:5740 STMicroelectronics STM32F407
```

In this example, I only had one device. Don't be worried about the STM32F407 designation. This is just the description given to the device ID 0483:5740 that ST Microelectronics registered. But how do you find out what device path to use? Try the following after plugging in your cable:

```
$ dmesg | grep 'USB ACM device'
[  709.468447] cdc_acm 2-7:1.0: ttyACM0: USB ACM device
```

This is obviously not very user friendly, but from this you find that the device name is /dev/ttyACM0. Listing it confirms this:

```
$ ls -l /dev/ttyACM0
crw-rw---- 1 root dialout 166, 0 Jan 25 23:38 /dev/ttyACM0
```

The next problem is having permissions to use the device. Notice that the group for the device is dialout. Add yourself to the dialout group (substitute fred with your own user ID):

```
$ sudo usermod -a -G dialout fred
```

Log out and log in again to verify that you have the correct group:

```
$ id
uid=1000(fred) gid=1000(fred) groups=1000(fred),20(dialout),24(cdrom),...
```

Being a member of the dialout group saves you from having to use root access to access the serial device.

MacOS USB Serial Device

Perhaps the simplest way to find the USB device under MacOS is to simply list the callout devices:

```
$ ls -l /dev/cu.*
crw-rw-rw- 1 root wheel 35,  1  6 Jan 15:14 /dev/cu.Bluetooth-Incoming-Port
crw-rw-rw- 1 root wheel 35,  3  6 Jan 15:14 /dev/cu.FredsiPhone-Wireless
crw-rw-rw- 1 root wheel 35, 45 26 Jan 00:01 /dev/cu.usbmodemFD12411
```

For the USB demo, the new device will appear as something like the path /dev/cu.usbmodemFD12411. The device number may vary, so look for cu.usbmodem in the pathname. Notice that all permissions are given.

Windows USB Serial Device

Serial devices under Windows show up as COM devices in the Device Manager once the cable is plugged in and the driver is installed. Figure 7-5 is an example screenshot.
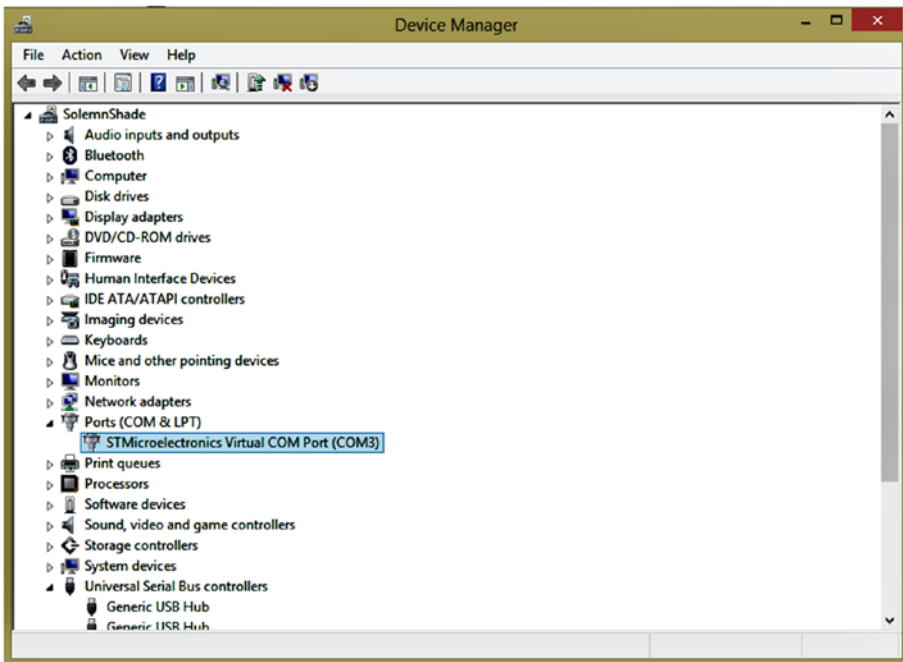
Figure 7-5. *Example Windows Device Manager dialog*

In this example, the USB device is attached as Windows port COM3. If you're using Cygwin under Windows, the device pathname is `/dev/ttyS2` (subtract 1 from the COM port number).

USB GPIO

The STMF103 series only supports USB on GPIO pins PA11 (`USB_DM`) and PA12 (`USB_DP`). There are *no alternate configurations* for USB. Further, there is no need to configure PA11 and PA12, because these are automatically taken over when the USB peripheral is enabled.[1] This is the only peripheral that I am aware of that behaves this way and is a tiny detail hidden in the reference manual RM0008 about alternate configurations. You *do*, however, need to enable the clocks for GPIOA and the USB peripheral.

Demo Source Code

Before running the supplied demo software, let's examine some of the USB-related portions of code found in the directory (again):

```
$ cd ~/stm32f103c8t6/rtos/usbcdcdemo
```

The code that will be discussed is found in source module usbcdc.c. Listing 7-1 illustrates the initialization code for the USB peripheral, using the libopencm3 driver and FreeRTOS for data queues.

Listing 7-1. The usb_start() Function for Initializing USB

```
0386: void
0387: usb_start(void) {
0388:    usbd_device *udev = 0;
0389:
0390:    usb_txq = xQueueCreate(128,sizeof(char));
0391:    usb_rxq = xQueueCreate(128,sizeof(char));
0392:
0393:    rcc_periph_clock_enable(RCC_GPIOA);
0394:    rcc_periph_clock_enable(RCC_USB);
0395:
0396:    // PA11=USB_DM, PA12=USB_DP
0397:    udev = usbd_init(&st_usbfs_v1_usb_driver,&dev,&config,
0398:        usb_strings,3,
0399:        usbd_control_buffer,sizeof(usbd_control_buffer));
0400:
0401:    usbd_register_set_config_callback(udev,cdcacm_set_config);
0402:
0403:    xTaskCreate(usb_task,"USB",200,udev,configMAX_PRIORITIES-1,NULL);
0404: }
```

Lines 390 and 391 create FreeRTOS queues, which will be used to communicate to and from the USB stream, respectively.

Since enabling the USB peripheral automatically takes over the GPIOs PA11 and PA12, all we have to do is enable the GPIO and USB clocks in lines 393 and 394. After that is done, the libopencm3 routine `usbd_init()` performs the rest in lines 397 to 399.

Once the peripheral is initialized, the callback `cdcacm_set_config()` is registered in line 401. Finally, a FreeRTOS task is created in line 403 to service the USB events.

cdcacm_set_config()

When the USB peripheral is contacted by the host controller, it will call upon the callback illustrated in Listing 7-2 to configure/reconfigure the USB CDC device.

Listing 7-2. The cdcadm_set_config() Callback

```
0030: // True when USB configured:
0031: static volatile bool initialized = false;
...
0252: static void
0253: cdcacm_set_config(
0254:   usbd_device *usbd_dev,
0255:   uint16_t wValue __attribute__((unused))
0256: ) {
0257:
0258:   usbd_ep_setup(usbd_dev,
0259:       0x01,
0260:       USB_ENDPOINT_ATTR_BULK,
0261:       64,
0262:       cdcacm_data_rx_cb);
0263:   usbd_ep_setup(usbd_dev,
0264:       0x82,
0265:       USB_ENDPOINT_ATTR_BULK,
0266:       64,
0267:       NULL);
```

```
0268:    usbd_register_control_callback(
0269:        usbd_dev,
0270:        USB_REQ_TYPE_CLASS | USB_REQ_TYPE_INTERFACE,
0271:        USB_REQ_TYPE_TYPE | USB_REQ_TYPE_RECIPIENT,
0272:        cdcacm_control_request);
0273:
0274:    initialized = true;
0275: }
```

From lines 258 to 262, it can be seen that callback cdcacm_data_rx_cb() is registered so that it can receive data. From the host's perspective, this is an OUT port, thus specified as endpoint 0x01 (OUT endpoint 1).

Next, lines 263 to 267 register another endpoint, which is considered as an IN port from the host controller's perspective. Hence, the IN endpoint 2 is specified with the high bit on in the constant 0x82.

Finally, control requests will call upon callback cdcacm_control_request() as registered in lines 268 to 272.

Lastly, the Boolean variable initialized is set to true in line 274 so that other tasks can know the ready status of the USB infrastructure.

cdc_control_request()

The USB infrastructure uses the cdcacm_control_request() callback to act on specialized messages (Listing 7-3). This driver reacts to two req->bRequest message types, the first of which is to satisfy a Linux deficiency (lines 203 to 209).

Listing 7-3. The cdcacm_control_request() Callback

```
0190: static int
0191: cdcacm_control_request(
0192:    usbd_device *usbd_dev __attribute__((unused)),
0193:    struct usb_setup_data *req,
0194:    uint8_t **buf __attribute__((unused)),
0195:    uint16_t *len,
```

```
0196:   void (**complete)(
0197:     usbd_device *usbd_dev,
0198:     struct usb_setup_data *req
0199:   ) __attribute__((unused))
0200: ) {
0201:
0202:   switch (req->bRequest) {
0203:   case USB_CDC_REQ_SET_CONTROL_LINE_STATE:
0204:       /*
0205:        * The Linux cdc_acm driver requires this to be implemented
0206:        * even though it's optional in the CDC spec, and we don't
0207:        * advertise it in the ACM functional descriptor.
0208:        */
0209:       return 1;
0210:   case USB_CDC_REQ_SET_LINE_CODING:
0211:       if ( *len < sizeof(struct usb_cdc_line_coding) ) {
0212:           return 0;
0213:       }
0214:       return 1;
0215:   }
0216:   return 0;
0217: }
```

Lines 210 to 214 check on the length of a structure and return fail if the length is out of line (line 212). Otherwise, a return of 1 indicates a "handled" status (line 214).

cdcacm_data_rx_cb()

This callback is invoked by the USB infrastructure when data has been sent over the bus to the STM32 MCU. The first thing performed in line 228 is to determine how much buffer space is remaining assigned to variable rx_avail. If there is insufficient space available, the callback simply returns in line 233. The host will send the same data again, later.

If we have room for some data, we decide how much in line 236. The call to usbd_ep_read_packet() in line 239 then obtains some or all of the received data. Lines 241 to 244 send it to the receive queue for the receiving task. See Listing 7-4.

Listing 7-4. The USB Receive Callback

```
0222: static void
0223: cdcacm_data_rx_cb(
0224:    usbd_device *usbd_dev,
0225:    uint8_t ep __attribute__((unused))
0226: ) {
0227:    // How much queue capacity left?
0228:    unsigned rx_avail = uxQueueSpacesAvailable(usb_rxq);
0229:    char buf[64];    // rx buffer
0230:    int len, x;
0231:
0232:    if ( rx_avail <= 0 )
0233:        return;       // No space to rx
0234:
0235:    // Bytes to read
0236:    len = sizeof buf < rx_avail ? sizeof buf : rx_avail;
0237:
0238:    // Read what we can, leave the rest:
0239:    len = usbd_ep_read_packet(usbd_dev,0x01,buf,len);
0240:
0241:    for ( x=0; x<len; ++x ) {
0242:        // Send data to the rx queue
0243:        xQueueSend(usb_rxq,&buf[x],0);
0244:    }
0245: }
```

USB Task

The task that we created for the USB handling is a forever loop starting in line 284. The loop must call the libopencm3 driver routine usbd_poll() frequently enough that the USB link is maintained by the host. This is done at the top of the loop in line 285 of Listing 7-5.

Listing 7-5. The usb_task() Function

```
0278: static void
0279: usb_task(void *arg) {
0280:    usbd_device *udev = (usbd_device *)arg;
0281:    char txbuf[32];
0282:    unsigned txlen = 0;
0283:
0284:    for (;;) {
0285:        usbd_poll(udev); /* Allow driver to do its thing */
0286:        if ( initialized ) {
0287:            while ( txlen < sizeof txbuf
0288:                && xQueueReceive(usb_txq,&txbuf[txlen],0)
0                        == pdPASS )
0289:                ++txlen; /* Read data to be sent */
0290:            if ( txlen > 0 ) {
0291:                if ( usbd_ep_write_packet(udev,0x82,
0                                txbuf,txlen) != 0 )
0292:                    txlen = 0; /* Reset if sent ok */
0293:            } else  {
0294:                taskYIELD(); /* Then give up CPU */
0295:            }
0296:        }
0297:    }
0298: }
```

The volatile bool variable initialized is checked in line 286. Until initialized is true, other USB calls like usbd_ep_write_packet() must be avoided.

After the driver has initialized, a check of the transmit queue is made in lines 287 to 289. As many queued characters as possible are taken from the queue to be sent. The sending of the USB data occurs in lines 290 to 292. If there are no characters to transmit, the FreeRTOS call to taskYIELD() is made to give another task CPU time.

From this, you can see that the purpose of this task is simply to send any queued bytes of data to the USB host. The receiving of data occurs from another place.

USB Receiving

When the application wants to read serial data, it calls upon usb_getc() or wrapper routines like usb_getline(). Listing 7-6 illustrates the code for usb_getc().

In line 367 you can see that it calls upon xQueueReceive() to pull a byte of received data from the queue. If there is no data, the call will block there because of the parameter given as portMAX_DELAY. Once the callback cdcacm_data_rx_cb() is invoked and queues up data, this code will receive data and unblock.

While it should never happen, the return of -1 in line 369 is taken if the queue has been destroyed or otherwise has become non-functional. Normally, the single character is returned by line 370.

Listing 7-6. The Listing of Function usb_getc()

```
0362: int
0363: usb_getc(void) {
0364:    char ch;
0365:    uint32_t rc;
0366:
0367:    rc = xQueueReceive(usb_rxq,&ch,portMAX_DELAY);
0368:    if ( rc != pdPASS )
0369:        return -1;
0370:    return ch;
0371: }
```

USB Sending

To send a byte of data to USB, it is put into the FreeRTOS usb_txq by function usb_putc(), as shown in Listing 7-7. Before it does that, however, a check is made in line 307 to make sure that the USB driver is ready. If it is not available yet, taskYIELD() is called in line 308 to share the CPU cycles.

Once the USB driver is known to be ready, the byte is queued in line 312, where it will block if the queue is full. Once bytes are drained from that queue, the character is queued and the call returns.

Listing 7-7. Sending Data Through USB Using Function usb_putc()

```
0303: void
0304: usb_putc(char ch) {
0305:   static const char cr = '\r';
0306:
0307:   while ( !usb_ready() )
0308:       taskYIELD();
0309:
0310:   if ( ch == '\n' )
0311:       xQueueSend(usb_txq,&cr,portMAX_DELAY);
0312:   xQueueSend(usb_txq,&ch,portMAX_DELAY);
0313: }
...
0407: bool
0408: usb_ready(void) {
0409:   return initialized;
0410: }
```

To make things character friendly, the function usb_putc() checks to see if you are
sending a \n (newline, also known as linefeed) character. If so, line 311 first sends a
carriage-return character. Under Unix/Linux, this type of processing is known as *cooked
mode*. The receiving side in the terminal emulator will then move the cursor to the start
of the line before advancing to the next line because of the newline.

USB Serial Demo

To demonstrate serial I/O over USB, I've modified an open source text-based game
written by Jeff Tranter. His source code, found in the module adventure.c, has been
modified to use the USB routines that have just been covered.

To build the code to be flashed, perform the following:

```
$ make clobber
$ make
$ make flash
```

After flashing the MCU, gently push the USB cable into the STM32 and connect the other end of the cable to your laptop/PC. Assuming you know the device name (from earlier in the chapter), set up your minicom or other terminal program (review minicom instructions in Chapter 6 if necessary). I recommend you save these settings to a profile name like "usb2" since they differ from the USB settings used later in this book.

With everything ready, start your terminal emulator as follows:

```
$ minicom usb2
Welcome to minicom 2.7

OPTIONS:
Compiled on Sep 17 2016, 05:53:15.
Port /dev/cu.usbmodemFD12411, 16:56:26

Press Meta-Z for help on special keys
...

    Abandoned Farmhouse Adventure
          By Jeff Tranter

Your three-year-old grandson has gone
missing and was last seen headed in the
direction of the abandoned family farm.
It's a dangerous place to play. You
have to find him before he gets hurt,
and it will be getting dark soon...
?
```

Don't worry if you missed the introductory text in the session shown (you can obviously read it here or shut down minicom and start over). This can happen if you had to mess around with the configuration of minicom. Entering "help" will get you the important information you need.

From the first screen, you can read about the adventure. Information is available by typing "help":

```
? help
Valid commands:
go east/west/north/south/up/down
look
```

```
use <object>
examine <object>
take <object>
drop <object>
inventory
help
You can abbreviate commands and
directions to the first letter.
Type just the first letter of
a direction to move.
?
```

The game consists of using a verb and sometimes an object. The following session gives you a sample:

```
? look
You are in the driveway near your car.
You see:
  key
You can go: north
? take key
Took key.
? inventory
You are carrying:
  flashlight
  key
?
```

Summary

USB is a large subject because it must adapt to many different uses. The serial stream shown in this chapter is one of the many applications of USB. Additionally, control structures were declared but left undescribed from the source module `usbcdc.c`. The interested reader is encouraged to study them and experiment with the source code. Several books have been written about USB, and this project gives you a foundation from which to start.

You have also seen how a convenient USB interface can be constructed between the STM32 and your laptop/PC. No baud rates, data bits, stop bits, parity, or flow control were required for you to configure the USB. Provided that the necessary driver support is present on the USB host, it is as simple as plugging in your cable.

While the focus has been on USB as a serial communications medium, the demo also highlighted some FreeRTOS facilities, like tasks and message queues. Having separately executing tasks and safe inter-task communications greatly simplifies application development.

Finally, the known USB defect of the Blue Pill is actually not that difficult to correct. Given the power of the STM32 MCU, available at the price of an AVR device, there is no reason for anyone to miss out on the fun!

Bibliography

1. Reference Manual RM0008, http://www.st.com/resource/en/reference_manual/cd00171190.pdf, Table 29, page 167.

EXERCISES

1. What GPIO preparation is necessary before enabling the USB peripheral?

2. What are the alternate GPIO configurations available for USB?

3. What libopencm3 routine must be called regularly to handle USB events?

SPI Flash

As resourceful as the STM32 MCU is, sometimes you need additional persistent data storage. Small applications may leave leftover program flash storage that can be utilized, but if you are collecting larger amounts of data, you will probably look to a serial flash solution.

This chapter will describe communication with the Winbond W25Q32 or W25Q64 chips using the SPI peripheral in master mode. The W25Q32 chip provides 4 MB of erasable flash storage, while the W25Q64 provides 8 MB. These chips can be purchased on eBay for a few dollars each, making them attractive for many applications.

Introducing W25QXX

The W25Q32/64 chips provide a fair amount of storage but require only a few wires to communicate. They operate from 2.7 to 3.6 volts, use 50 µA of standby current, and use approximately 15 mA for data reads. Writing and erasure require a little more at 25 mA. Furthermore, the W25QXX chips can be powered down under software control to save power when you need to.

Since these flash chips use the SPI bus to communicate, let's briefly review how SPI operates.

Serial Peripheral Interface Bus

The serial peripheral interface (SPI) is a synchronous serial interface that communicates over short distances using three wires and a chip-select signal. One end of the bus operates as the master on the bus while the remaining devices are slave devices. The SPI interface was developed by Motorola in the late 1980s and has since become

a de facto standard.[1] Figure 8-1 illustrates one master communicating with one slave device. This is how this chapter's demo project will be configured. Additional slave devices could be attached to the bus, but each would have its own chip-select signal.
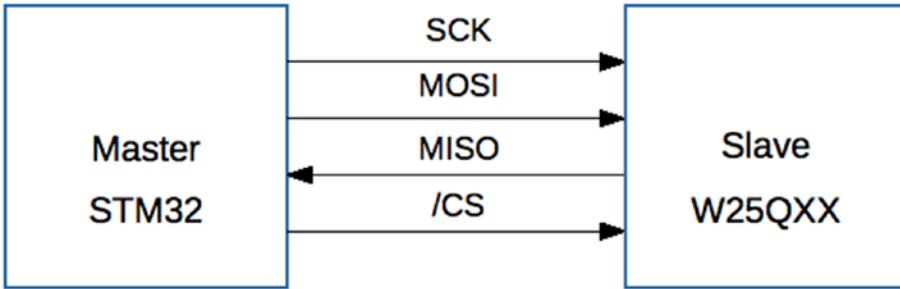
Figure 8-1. *SPI single master to single slave example*

The *system clock* signal (SCK) line provides clock pulses that time the data bits being transmitted and received. Signal MOSI is the *master out slave in* data, while MISO is the *master in slave out* signal. The fourth signal is the device *chip select* $\left(\overline{CS} \right)$, which is used to activate the chosen device. It is shown with an overhead bar or preceding slash (/CS) to indicate that it is active in the low state. Sometimes this signal is referred as the *slave select* $\left(\overline{SS} \right)$.

One of the unique aspects of the SPI bus is its method of communication. As the master sends out data bits on the MOSI line, the slave is simultaneously returning data bits to the master on the MISO line. Figure 8-2 illustrates how the pair behaves as two sets of shift registers.
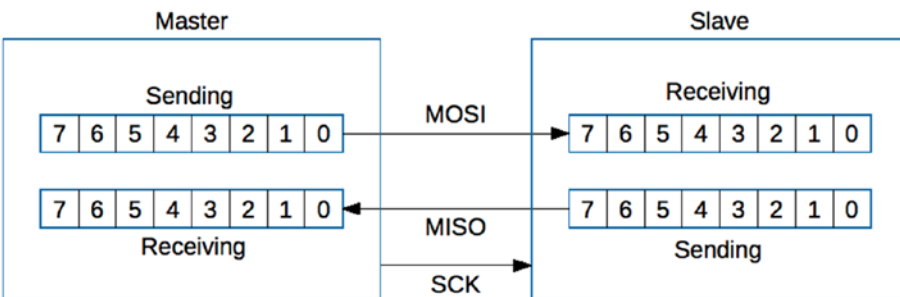
Figure 8-2. *SPI master and slave as a set of shift registers*

The SPI master always generates the clock pulse (SCK) to time when the data is sampled and shifted into the receiving register. Once the full word length is received, the receiving slave and master can simultaneously read the received data.

The SPI bus design leads to some quirky programming. For example, the slave device may not know what data to send until it has received a command word from the master. Consequently, when the master sends the command word to the slave, the first word received from the slave is discarded because it is meaningless. Once the slave device has received the command word it then knows how to reply. But the slave needs the master to send a dummy word to allow its reply to be shifted into the master's receive register. Because of this characteristic, SPI programming often requires the discarding of some received data and the sending of dummy words. The word size for the STM32 SPI controller can be 8 or 16 bits in length.

Chip Select

You might be asking "Why do we need a chip-select line when there is only one slave involved in this demo?" The problem is that there can be bus line noise. To guard against that, the slave needs to know when a transmission begins and ends. If noise is received on the SCK line, for example, the slave could end up one bit out of step with the master. Perhaps a scrambled command could be received by the flash chip as a "chip erase" function, which would be disastrous. For this reason, the /CS goes low prior to the first bit of data being sent by the master. This tells the slave device that the *first* bit is coming. When the last word of data has been sent, the /CS returns to the high state to signal the end of the transmission. The Winbond flash chip will insist upon this prior to executing a write or erase operation; otherwise, the command is disregarded.

Wiring and Voltages

When wiring up UARTs, it is often necessary to connect TX to RX, and RX to TX, and so forth, depending upon the sense of the device and how the manufacturer labeled the connections. This can be confusing. With the SPI bus, the situation is very simple—the SCK line *always* connects to SCK, MOSI *always* to MOSI, MISO *always* to MISO and $\overline{CS}$ to $\overline{CS}$.

The SPI bus voltage can vary, being usually 5 volts or 3.3 volts. The Winbond W25QXX devices can operate at the 3.3-volt level, making it simple to interface with the STM32.

SPI Circuit

Figure 8-1 illustrates the full circuit related to our SPI flash project. The Winbond chip includes some other pins that we haven't discussed, as follows:

- /WP Write Protect (wire to +3.3 V to enable writes)

- /HOLD Hold Input (wire to +3.3 V when not used)

Both of these features are not used in this chapter and should be wired to the +3.3-volt supply. The /WP signal is a safety option that you might find useful in some applications. When /WP is grounded, no writes or erasures are possible.

Hardware /NSS Control

A feature of the STM32 SPI peripheral that has vexed a number of people, judging by forum posts, is the optional *hardware* drive of the /CS pin in SPI master mode. If you omit the pull-up resistor R1 shown in Figure 8-3, you will discover that it doesn't work. Many have reported that "it doesn't seem to work" or "it just doesn't seem to do anything." The forums' answer to this problem has been to advise the use of *software* management of the pin instead (operate as a GPIO).
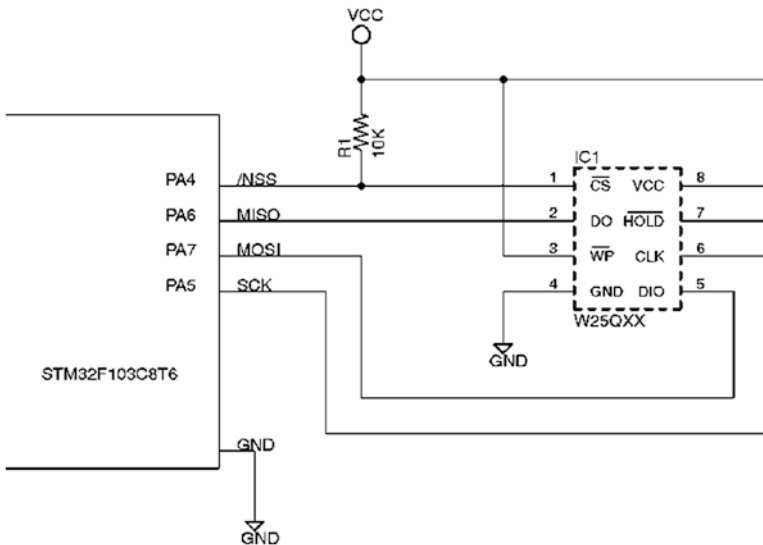
Figure 8-3. *STM32 wired up to the W25Q32 or W25Q64. If you have a PCB with a different layout, ignore the PCB's pin numbers and match by function (CLK for example, is connected to SCK on the MCU).*

This problem is partly based upon the *assumption* that the /NSS (/CS) pin is a totem-pole (push/pull) output. After all, that is how it is usually configured during SPI setup (see line 652 of Listing 8-2 later). The ST documentation is also weak on this point. The only hint at this behavior is in the reference manual (RM0008), section "25.3 SPI functional description" under "Slave select (NSS) management":

> *NSS signal is driven low when the master starts the communication and is kept low until the SPI is disabled.*

The documentation never mentions the /NSS signal being driven high. Hmmm. Then, there is Application Note AN2576, which describes "STM32F10xxx SPI and M25P64 Flash memory communication." Present in that document's Figure 5 is a 10-kohm pull-up resistor that is never mentioned.

This characteristic of the SPI peripheral is not entirely surprising when you read about the features of the peripheral. One of the features touted is support for *multi-master* mode. In this mode of operation, the /NSS pin would have to function as both an input *and* output. An open-drain driver is suited for this mode of operation. In a perfect world, the peripheral would enable push/pull output in *single* master mode and use open drain for multi-master mode. But this is not the case here. Reading datasheets and working with hardware often leads to some interesting puzzles.

If you're just getting started in digital electronics, then the simple answer in this circuit is that you need that 10-kohm resistor. Perhaps some readers may be muttering, "Why all this fuss about hardware control? Why not just control /NSS with GPIO commands?" That word "just" creates so much trouble!

From a purely logical point of view, and disregarding the small efficiency loss, GPIO control of the /NSS is perfectly valid, if a nuisance to code for. But the main reason for desiring the hardware /NSS pin control is to reduce the chance of noise corrupting SPI messages. The timespan between starting an SPI transaction and activating the /NSS line is shorter in hardware than when setting the GPIO in software. Likewise, the SPI hardware can deactivate the /NSS line at the end of the transaction sooner than a software GPIO action can. The times aren't majorly different, but it does reduce the opportunity for message corruption.

Not to flog a dead horse further, one final reason for employing hardware control of the /NSS line is that it saves us from having to do it ourselves. That may seem like a Captain Obvious thing to say, but it means that we can't forget to disable the /NSS line. If we were to forget, the last write or erase operation would be ignored by the flash chip. The worst and most insidious errors are those that go unnoticed until it becomes too late to trace why.

Note that V_{cc} here is +3.3-volt supply. Signals /WP and /HOLD are active low and must be wired to V_{cc} to disable them.

Figure 8-4 illustrates the two main packages that the W25Q32 comes in. The price for the DIP (Dual Inline Package) package is about the same as for the SOIC (Small Outline Integrated Circuit) on a PCB, from eBay.
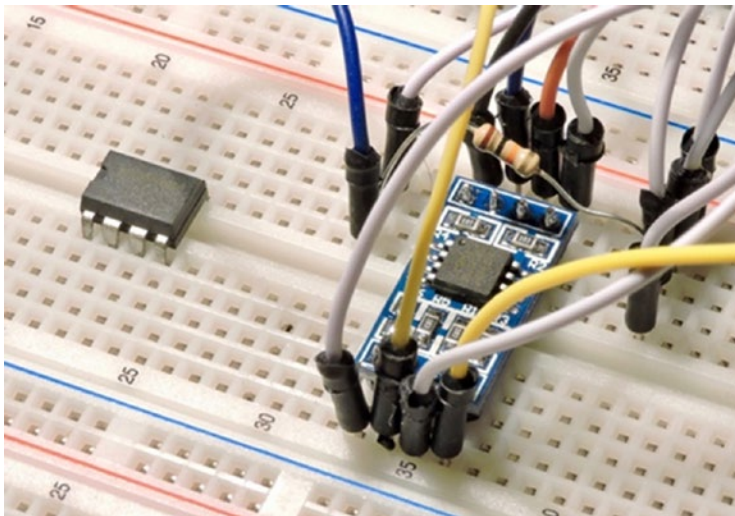
Figure 8-4. *W25Q32 in DIP form (left) and W25Q32 as SOIC on PCB (right)*

STM32 SPI Configuration

To communicate with the external flash chip, we need to configure and ready the STM32 SPI peripheral. This chapter's demo source code is found in the following directory:

```
$ cd ~/stm32f103c8t6/rtos/winbond
```

Table 8-1 summarizes the GPIO pins that will be used to connect the SPI1 peripheral to the Winbond flash chip. These are also shown in schematic in Figure 8-3.

Table 8-1. *STM32 SPI1 Pins Used*

GPIO Pin	SPI Function	Description
PA4	/CS	Chip Select (active low)
PA5	SCK	System Clock
PA6	MISO	Master In, Slave Out
PA7	MOSI	Master Out, Slave In

Listing 8-1 provides the general initialization source code found in the main program. Line 679 enables the clock for GPIOA, since our SPI peripheral is using those pins (Table 8-1). The remaining SPI setup is performed in line 685, function spi_setup(), which we'll examine shortly.

Listing 8-1. The Main Program Initialization

```
0674: int
0675: main(void) {
0676:
0677:    rcc_clock_setup_in_hse_8mhz_out_72mhz(); // Blue pill
0678:
0679:    rcc_periph_clock_enable(RCC_GPIOA);
0680:    rcc_periph_clock_enable(RCC_GPIOC);
0681:
0682:    // LED on PC13
0683:    gpio_set_mode(GPIOC,GPIO_MODE_OUTPUT_2_MHZ,
            GPIO_CNF_OUTPUT_PUSHPULL,GPIO13);
0684:
0685:    spi_setup();
0686:    gpio_set(GPIOC,GPIO13);    // PC13 = on
0687:
0688:    usb_start(1);
0689:    std_set_device(mcu_usb);   // Use USB for std I/O
0690:    gpio_clear(GPIOC,GPIO13);  // PC13 = off
0691:
```

```
0692:   xTaskCreate(monitor_task,"monitor",
            500,NULL,configMAX_PRIORITIES-1,NULL);
0693:   vTaskStartScheduler();
0694:   for (;;);
0695:   return 0;
0696: }
```

To allow us to focus on SPI in this chapter, we use a library to furnish the USB communications to the desktop. The static library is located here:

- ~/stm32f103c8t6/rtos/libwwg/libwwg.a

The source code for the library is found in the following two directories:

- ~/stm32f103c8t6/rtos/libwwg/include

- ~/stm32f103c8t6/rtos/libwwg/src

Lines 688 and 689 perform the USB initialization, allowing the program to communicate with a terminal program. Line 689 simply redirects all calls to std_printf() to usb_printf() instead, and so forth. If you should later decide to use a UART for communication, this redirector can be set for that instead.

Listing 8-2 shows the spi_setup() routine, which covers the SPI specifics. The following steps are used to initialize peripheral SPI1:

1. The clock for SPI1 is enabled (line 648)

2. GPIOA pins are configured for (lines 649–654):

    a. alternate function output (push-pull)

    b. 50 MHz (for fast rise/fall times)

    c. for PA4, PA5, and PA7

3. GPIO PA6 is configured for input without pull-up resistor (lines 655–660)

4. The SPI1 peripheral is reset (line 661)

5. SPI1 is configured to use:

    a. a fpclk divisor 256 (line 664)

    b. SCK polarity of 0 (low) when idle (line 655)

    c. Clock phase occurs on first transition (line 666)

     d.   Word length is 8 bits (line 667)

     e.   Bits are shifted out MSB (Most Significant Bit) first (line 668)

6.   SPI1 is using hardware /CS management (i.e., *not* using software slave management, line 670).

7.   SPI peripheral can assert /CS (line 671).

Listing 8-2. SPI Peripheral Setup

```
0645: static void
0646: spi_setup(void) {
0647:
0648:   rcc_periph_clock_enable(RCC_SPI1);
0649:   gpio_set_mode(
0650:       GPIOA,
0651:           GPIO_MODE_OUTPUT_50_MHZ,
0652:           GPIO_CNF_OUTPUT_ALTFN_PUSHPULL,
0653:           GPIO4|GPIO5|GPIO7  // NSS=PA4,SCK=PA5,MOSI=PA7
0654:   );
0655:   gpio_set_mode(
0656:       GPIOA,
0657:       GPIO_MODE_INPUT,
0658:       GPIO_CNF_INPUT_FLOAT,
0659:       GPIO6                  // MISO=PA6
0660:   );
0661:   spi_reset(SPI1);
0662:   spi_init_master(
0663:           SPI1,
0664:           SPI_CR1_BAUDRATE_FPCLK_DIV_256,
0665:           SPI_CR1_CPOL_CLK_TO_0_WHEN_IDLE,
0666:           SPI_CR1_CPHA_CLK_TRANSITION_1,
0667:           SPI_CR1_DFF_8BIT,
0668:           SPI_CR1_MSBFIRST
0669:   );
0670:   spi_disable_software_slave_management(SPI1);
0671:   spi_enable_ss_output(SPI1);
0672: }
```

SPI Clock Rate

One of the most vexing things about the STM32 platform is the complexity of the clock system. The question we want to answer is what frequency does the macro `SPI_CR1_BAUDRATE_FPCLK_DIV_256` provide? Part of the answer lies in the determination of f_{PCLK} For the STM32F103, SPI1 uses the APB2 bus clock, which has a maximum frequency of 72 MHz. SPI2 uses the APB1 bus clock, which has a maximum frequency of 36 MHz.

The main program used the following libopencm3 function to establish some of the main clocks:

```
0677:   rcc_clock_setup_in_hse_8mhz_out_72mhz(); // Blue pill
```

With these clock settings in effect, we can summarize the SPI's f_{PCLK} as in Table 8-2.

Table 8-2. *Frequencies for SPI1 and SPI2, Assuming rcc_clock_setup_in_hse_8mhz_out_72mhz( )*

Clock	Bus	Peripheral	fPCLK
PCLK1	APB1	SPI2	36 MHz
PCLK2	APB2	SPI1	72 MHz

With this information, we can summarize the choices for the SPIx clock divisors, as shown in Table 8-3. I verified with a DSO (digital storage oscilloscope) that the period of the SCK signal is about 3.56 µS when running the demonstration program. This evaluates to a frequency of about 281 kHz, as expected.

Table 8-3. *SPI Divisor Frequencies, Based Upon Table 8-3.*

Divisor	Macro	SPI1 Frequency	SPI2 Frequency
2	`SPI_CR1_BAUDRATE_FPCLK_DIV_2`	36 MHz	18 MHz
4	`SPI_CR1_BAUDRATE_FPCLK_DIV_4`	18 MHz	9 MHz
8	`SPI_CR1_BAUDRATE_FPCLK_DIV_8`	9 MHz	4.5 MHz
16	`SPI_CR1_BAUDRATE_FPCLK_DIV_16`	4.5 MHz	2.25 MHz
32	`SPI_CR1_BAUDRATE_FPCLK_DIV_32`	2.25 MHz	1.125 MHz
64	`SPI_CR1_BAUDRATE_FPCLK_DIV_64`	1.125 MHz	562.5 kHz
128	`SPI_CR1_BAUDRATE_FPCLK_DIV_128`	562.5 kHz	281.25 kHz
256	`SPI_CR1_BAUDRATE_FPCLK_DIV_256`	281.25 kHz	140.625 kHz

I chose a low frequency for this demonstration to guarantee good results on the breadboard. Sometimes with breadboards and long wires, noise can be disruptive to the SPI communication. With the source code at your disposal, you might try higher bit rates after your initial success. The Winbond chip will read continuously up to 50 MHz, but SPI1 is limited to 36 MHz on the STM32 platform, establishing your upper limit.

SPI Clock Modes

The spi_setup() routine in Listing 8-2 used the following configuration parameter:

```
0665:           SPI_CR1_CPOL_CLK_TO_0_WHEN_IDLE,
```

What does that mean to the programmer?

SPI can operate in one of four modes, which can lead to confusion. The Winbond flash chips used in this chapter can operate in modes 0 or 3. Figure 8-5 illustrates the relationships between the various libopencm3 configuration macros. It must be kept in mind that clock polarity and phase *together* determine the SPI mode of operation.
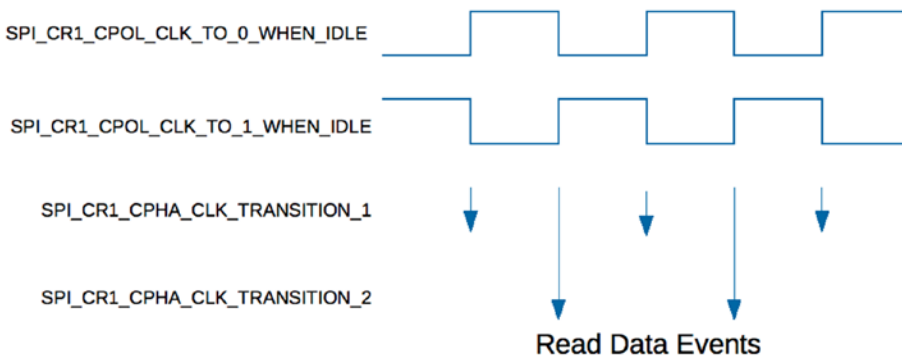
Figure 8-5. *Clock polarity and phase configurations*

When `SPI_CR1_CPOL_CLK_TO_0_WHEN_IDLE` is used with `SPI_CR1_CPHA_CLK_TRANSITION_1`, the receiver samples the data input at the rising clock transition (short arrows of Figure 8-5). If the same clock polarity is used and `SPI_CR1_CPHA_CLK_TRANSITION_2` is configured instead, the data is sampled at the falling edge of the clock (long arrows).

The situation is reversed when `SPI_CR1_CPOL_CLK_TO_1_WHEN_IDLE` polarity is used (second line from the top in Figure 8-5). The falling edge (short arrows) of the clock are used when `SPI_CR1_CPHA_CLK_TRANSITION_1` is configured; otherwise, the rising clock (long arrows) are used.

Table 8-4 summarizes the SPI modes using the libopencm3 macro names. Knowing that the Winbond W25QXX chips will operate on modes 0 or 3, we can arrive at the conclusion that they operate only on the rising edge of the SCK signal.

Table 8-4. *A Summary of SPI Modes by Number*

SPI Mode	Clock Polarity	Clock Phase
0	`SPI_CR1_CPOL_CLK_TO_0_WHEN_IDLE`	`SPI_CR1_CPHA_CLK_TRANSITION_1`
1	`SPI_CR1_CPOL_CLK_TO_0_WHEN_IDLE`	`SPI_CR1_CPHA_CLK_TRANSITION_2`
2	`SPI_CR1_CPOL_CLK_TO_1_WHEN_IDLE`	`SPI_CR1_CPHA_CLK_TRANSITION_1`
3	`SPI_CR1_CPOL_CLK_TO_1_WHEN_IDLE`	`SPI_CR1_CPHA_CLK_TRANSITION_2`

Another point needs to be made about the SPI clock polarity, at least in reference to the libopencm3 macro names. The macro SPI_CR1_CPOL_CLK_TO_0_WHEN_IDLE describes the clock idle polarity *during chip-select time*. Figure 8-6 is a captured scope trace of SCK becoming active with the /CS going low. Prior to /CS activation, the clock was resting at the high level. But during the SPI transaction (/CS active), the clock was indeed idle at the low level. This is important to bear in mind when examining the SPI signals.

Figure 8-6. DSO trace of SCK and /CS, with SCK idle at low

Endianess and Word Length

Now, we can cover the final aspects of the SPI1 configuration (from Listing 8-2):

```
0662:   spi_init_master(
0663:         SPI1,
0664:         SPI_CR1_BAUDRATE_FPCLK_DIV_256,
0665:         SPI_CR1_CPOL_CLK_TO_0_WHEN_IDLE,
0666:         SPI_CR1_CPHA_CLK_TRANSITION_1,
0667:         SPI_CR1_DFF_8BIT,
0668:         SPI_CR1_MSBFIRST
0669:   );
```

Macro SPI_CR1_DFF_8BIT (line 667) specifies that our word length is a byte (8 bits). Macro SPI_CR1_MSBFIRST (line 668) indicates that we will be transmitting the most significant bit first (big endian). The other libopencm3 choices are SPI_CR1_DFF_16BIT and SPI_CR1_LSBFIRST.

The SPI setup routine ends with two more calls:

```
0670:   spi_disable_software_slave_management(SPI1);
0671:   spi_enable_ss_output(SPI1);
```

Line 670 simply indicates in a backhanded way that we will be using a hardware assertion of the NSS pin (/CS). This call may not be strictly necessary after a reset, but it does serve to document our intention. Line 671 indicates that the SPI peripheral is to assert control of the NSS (/CS) pin. With those steps performed, SPI1 is ready for use.

SPI I/O

With the SPI device configured, we can now initiate commands on the SPI bus and communicate with the Winbond W25Q32 or its larger cousin, W25Q64. No interrupts are used for this example because our code is the *master*. As the master SPI device, our code sets the timing of the transactions. The slave marches to our drummer. Consequently, if our code is held up for any reason, the slave will wait for us. If there is a risk of extremely long delays, you may want to use interrupts to avoid bus noise.

SPI can perform send-only, read-only, or bidirectional send and receive operations. In our application, the Winbond chip will be sending back data, so we will always use the spi_xfer() function so that we can both send and receive a byte. This will be illustrated in the next section.

Read SR1

Listing 8-3 illustrates the code found in module main.c, which performs the SPI read of Winbond device status register one (SR1).

Listing 8-3. The Read Status Routine w25_read_sr1( )

```
0059: static uint8_t
0060: w25_read_sr1(uint32_t spi) {
0061:   uint8_t sr1;
```

```
0062:
0063:    spi_enable(spi);
0064:    spi_xfer(spi,W25_CMD_READ_SR1);
0065:    sr1 = spi_xfer(spi,DUMMY);
0066:    spi_disable(spi);
0067:    return sr1;
0068: }
```

The process begins by enabling SPI1 in line 63 (SPI1 is passed as the argument spi). This causes the hardware to assert the /CS signal and enable the clock (SCK). Then, libopencm3 function spi_xfer() is called to send a byte, which is defined as a macro:

```
0036: #define W25_CMD_READ_SR1        0x05
```

As part of the transaction, spi_xfer() returns a byte from the flash device, which in this case is discarded (Line 64). The received value is discarded because the flash device doesn't know what to send us until it receives our command. This kind of thing happens frequently in SPI transactions.

Line 65 now sends a *dummy* value (zero) so that it can receive the flash device's answer now that it knows what we are asking for. This return value is saved in variable sr1.

Finally, the SPI1 device is disabled in line 66 to end the SPI transaction. At this point, the hardware deasserts /CS, releasing the SPI bus. The flash device now enters a standby state. The read value in variable sr1 is then returned to the caller.

Once the configuration of the SPI device is complete, the use of the peripheral is nice and simple.

Waiting for Ready

The Winbond "read status" command is the only command that the flash device accepts at *any* time. All other requests require that the device be "not busy." If the device is queried when it is busy, the request is silently ignored. This is a Winbond flash device feature and is not related to SPI.

Because the flash device might be busy writing a page or erasing sectors, it is convenient to use a function to query for ready status. Listing 8-4 shows the code used.

Listing 8-4. The Winbond Wait Ready Function

```
0081: static void
0082: w25_wait(uint32_t spi) {
0083:
0084:    while ( w25_read_sr1(spi) & W25_SR1_BUSY )
0085:        taskYIELD();
0086: }
```

The listed w25_wait() function queries the status register SR1 and then checks the bit W25_SR1_BUSY. If that bit is set, the routine calls upon taskYIELD() in line 85 to allow other tasks to enjoy use of the CPU. When that bit becomes zero, the routine simply returns.

Read Manufacturer ID

The "read manufacturer ID" command is interesting because of the interplay of ignored received data and dummy writes. Listing 8-5 illustrates.

Listing 8-5. The Read Manufacturer ID Function

```
0107: static uint16_t
0108: w25_manuf_device(uint32_t spi) {
0109:    uint16_t info;
0110:
0111:    w25_wait(spi);
0112:    spi_enable(spi);
0113:    spi_xfer(spi,W25_CMD_MANUF_DEVICE);  // Byte 1
0114:    spi_xfer(spi,DUMMY);                 // Dummy1 (2)
0115:    spi_xfer(spi,DUMMY);                 // Dummy2 (3)
0116:    spi_xfer(spi,0x00);                  // Byte 4
0117:    info = spi_xfer(spi,DUMMY) << 8;     // Byte 5
0118:    info |= spi_xfer(spi,DUMMY);         // Byte 6
0119:    spi_disable(spi);
0120:    return info;
0121: }
```

Notice how line 111 calls upon `w25_wait()` to block until the flash device is ready for a new command. Then, the command to read the manufacturer device information is written to the SPI bus in line 113. The returned byte is discarded.

Lines 114 and 115 both send dummy bytes (zero was used). This is necessary to give the flash device enough clock pulses to read and ready the data to be sent back in response. Notice that those received bytes are also ignored (the flash device was not yet ready). Line 116 has the value `0x00` written out (as per flash device specs) to start the receiving of data on the next byte.

Finally, in lines 117 and 118, two more dummy bytes are sent out to cause the slave device to transmit its data. We save the values received in the 16-bit variable `info`, which is later returned in line 120.

This may seem like a crazy transaction, but device transactions are often this way.

Writing Flash

The Winbond chips are very careful to protect your flash storage. They offer an extensive API for protecting regions of the supported memory, all of it by software, or the entire chip by the assertion of the /WP signal on the chip. This makes sense in desktop motherboards where you don't want to lose BIOS code or settings.

The chip's flash safety has another consequence. After power up, after a data write or erase operation, the chip returns to a "write disabled" mode. To perform a data write, it must be preceded by a "write enable" operation. This is done by setting the "Write Enable Latch" option in the status register 1 (`SR1`). Listing 8-6 shows the code used for this purpose.

Listing 8-6. Enabling the Write Enable Latch

```
0095: static void
0096: w25_write_en(uint32_t spi,bool en) {
0097:
0098:   w25_wait(spi);
0099:
0100:   spi_enable(spi);
0101:   spi_xfer(spi,en ? W25_CMD_WRITE_EN : W25_CMD_WRITE_DI);
0102:   spi_disable(spi);
```

```
0103:
0104:   w25_wait(spi);
0105: }
```

Line 98 waits for the device to become ready (else any further commands would be ignored). Depending upon whether the routine was called to enable or disable writes, the appropriate command is sent in line 101. Since setting the latch might require some device time, we wait for a ready in line 104 prior to returning.

You should probably read the SR1 register back to see if that succeeded or not. If the /WP signal was set to active on the chip, then this operation would fail. In our project we have hardwired /WP to be inactive, so this was disregarded.

With the flash chip write enabled, we can now write one or more bytes of flash. Listing 8-7 presents the function w25_write_data() for programming data.

Listing 8-7. Write Data Function

```
0211: static unsigned         // New address is returned
0212: w25_write_data(uint32_t spi,uint32_t addr,void *data,uint32_t bytes)
{
0213:   uint8_t *udata = (uint8_t*)data;
0214:
0215:   w25_write_en(spi,true);
0216:   w25_wait(spi);
0217:
0218:   if ( w25_is_wprotect(spi) ) {
0219:       std_printf("Write disabled.\n");
0220:       return 0xFFFFFFFF;    // Indicate error
0221:   }
0222:
0223:   while ( bytes > 0 ) {
0224:       spi_enable(spi);
0225:       spi_xfer(spi,W25_CMD_WRITE_DATA);
0226:       spi_xfer(spi,addr >> 16);
0227:       spi_xfer(spi,(addr >> 8) & 0xFF);
0228:       spi_xfer(spi,addr & 0xFF);
```

```
0229:        while ( bytes > 0 ) {
0230:            spi_xfer(spi,*udata++);
0231:            --bytes;
0232:            if ( (++addr & 0xFF) == 0x00 )
0233:                break;
0234:        }
0235:        spi_disable(spi);
0236:
0237:        if ( bytes > 0 )
0238:            w25_write_en(spi,true); // More to write
0239:    }
0240:    return addr;
0241: }
```

Line 215 enables the "Write Enable Latch" option and checks that it got set in line 218 (by reading SR1). If the "Write Enable Latch" is not set, the console receives the message "Write disabled." in line 219, prior to returning a fail code.

The loop in lines 223 through 239 issues the "data write" command and three flash address bytes. Then, the bytes are transferred in the loop of lines 229 and 234. The new flash address is returned in the lower 24 bits of the return value.

The outer loop (Line 223) is designed to perform writes within 256-byte pages. The Winbond chip will wrap the address around within the same page if you try to cross page boundaries, so this code writes each page as a separate SPI command.

Flash Erase

It is easy to forget that we are dealing with flash memory. For values to be written successfully, the affected memory must be erased first. In the erased state, the byte has the value 0xFF (all bits set to 1). Once the byte is written as 0x00, it cannot be set back to 0xFF without an erase operation (nor can any bit be set back to 1).

Despite that, it is possible to cheat when writing a flash file system. Let's say you have one byte under consideration that represents eight clusters of data storage, where a 1-bit represents an available cluster and a 0-bit represents a cluster that is in use. If the present byte value is 0x7F, with the high bit (bit 7) cleared, you can allocate the next cluster by

zeroing the next bit without doing an erase. A write of value 0x3F (or even 0xBF) will clear the next bit, resulting in a read-back value of 0x3F. Notice that writing 0xBF also works, because the seventh bit is ignored when it is already zeroed. Programming 1-bits are no-ops, but programming 0-bits flip 1-bits to zeros.

Eventually, however, no matter how clever the scheme, the reality requires the eventual use of an erase operation. The difficulty in this is that an erase must be performed on a large-block basis. The W25QXX chips allow you to erase the following:

- One sector (4 KB)

- 32 KB block

- 64 KB block

- Entire chip

Listing 8-8 illustrates the chip-erase code used in the demo program. Write-protect status is checked in lines 170 to 173. If the chip is protected, a message of protest is given in line 171 prior to an error return. Lines 175 to 177 perform the chip erase, while the remainder of the function checks to see if the operation was successful. If the "Write Enable Latch" did not return to disabled, then this indicates that the command failed or was ignored. If this happens, there is likely a software problem or the message on the SPI bus was corrupted somehow.

Listing 8-8. Chip-Erase Function

```
0167: static bool
0168: w25_chip_erase(uint32_t spi) {
0169:
0170:   if ( w25_is_wprotect(spi) ) {
0171:       std_printf("Not Erased! Chip is not write enabled.\n");
0172:       return false;
0173:   }
0174:
0175:   spi_enable(spi);
0176:   spi_xfer(spi,W25_CMD_CHIP_ERASE);
0177:   spi_disable(spi);
0178:
```

```
0179:   std_printf("Erasing chip..\n");
0180:
0181:   if ( !w25_is_wprotect(spi) ) {
0182:       std_printf("Not Erased! Chip erase failed.\n");
0183:       return false;
0184:   }
0185:
0186:   std_printf("Chip erased!\n");
0187:   return true;
0188: }
```

The remaining erasure functions are nearly the same, except for the fact that they identify the block number that they are erasing. Listing 8-9 illustrates the routine used.

Listing 8-9. Block-Erasure Routine w25_erase_block( )

```
0233: static void
0234: w25_erase_block(uint32_t spi,uint32_t addr,uint8_t cmd) {
0235:   const char *what;
0236:
0237:   if ( w25_is_wprotect(spi) ) {
0238:       std_printf("Write protected. Erase not performed.\n");
0239:       return;
0240:   }
0241:
0242:   switch ( cmd ) {
0243:   case W25_CMD_ERA_SECTOR:
0244:       what = "sector";
0245:       addr &= ~(4*1024-1);
0246:       break;
0247:   case W25_CMD_ERA_32K:
0248:       what = "32K block";
0249:       addr &= ~(32*1024-1);
0250:       break;
```

```
0251:    case W25_CMD_ERA_64K:
0252:        what = "64K block";
0253:        addr &= ~(64*1024-1);
0254:        break;
0255:    default:
0256:        return;       // Should not happen
0257:    }
0258:
0259:    spi_enable(spi);
0260:    spi_xfer(spi,cmd);
0261:    spi_xfer(spi,addr >> 16);
0262:    spi_xfer(spi,(addr >> 8) & 0xFF);
0263:    spi_xfer(spi,addr & 0xFF);
0264:    spi_disable(spi);
0265:
0266:    std_printf("%s erased, starting at %06X\n",
0267:        what,(unsigned)addr);
0268: }
```

The argument passed as addr is taken to be the block number to be erased. The argument cmd then indicates which type of erasure to perform. Line 242 then determines what type of erasure is being performed and performs a mask operation on addr according to the block size being used.

The erasure command happens in lines 260 to 263, where the command and three bytes of block numbers are transmitted.

Reading Flash

Once the flash has been written, we need the ability to read it back. Listing 8-10 shows a function to perform this task.

Listing 8-10. A Function to Read SPI Flash

```
0191: static uint32_t       // New address is returned
0192: w25_read_data(uint32_t spi,uint32_t addr,void *data,uint32_t bytes) {
0193:   uint8_t *udata = (uint8_t*)data;
0194:
```

```
0195:    w25_wait(spi);
0196:
0197:    spi_enable(spi);
0198:    spi_xfer(spi,W25_CMD_FAST_READ);
0199:    spi_xfer(spi,addr >> 16);
0200:    spi_xfer(spi,(addr >> 8) & 0xFF);
0201:    spi_xfer(spi,addr & 0xFF);
0202:    spi_xfer(spi,DUMMY);
0203:
0204:    for ( ; bytes-- > 0; ++addr )
0205:        *udata++ = spi_xfer(spi,DUMMY);
0206:
0207:    spi_disable(spi);
0208:    return addr;
0209: }
```

The argument addr indicates the flash relative address to read, while arguments data and bytes indicate where to place the read data. Line 198 issues the SPI "read" command, and then three bytes of address information is transmitted to the slave device. The "Fast Read" requires a dummy byte be written out after the address (line 202). After that, the loop in lines 204 and 205 reads back the bytes transmitted by the flash device.

The "Fast Read" Winbond command was not used here for speed. The regular read command has the problem that it will wrap around within the current 256-byte page. To allow reads to cross page boundaries, the "Fast Read" command is used instead.

Demonstration

Enough tech talk! Time for a demonstration. If you've not already done so, build the program now (in directory ~/stm32f103c8t6/rtos/winbond):

```
$ make clobber
$ make
...
arm-none-eabi-size main.elf
   text    data    bss    dec    hex    filename
  17376    1472  18104  36952   9058    main.elf
```

Attach your programmer and perform the following:

```
$ make flash
/usr/local/bin/st-flash  write main.bin 0x8000000
st-flash 1.3.1-9-gc04df7f-dirty
2017-11-04T10:03:37 INFO src/usb.c: -- exit_dfu_mode
2017-11-04T10:03:37 INFO src/common.c: Loading device parameters....
...
2017-11-04T10:03:39 INFO src/common.c: Starting verification of write
  complete
2017-11-04T10:03:39 INFO src/common.c: Flash written and verified!
  jolly good!
```

With the STM32 device flashed, unplug the programmer first (important) and then plug in a USB cable between the STM32 and your desktop. I am using minicom as the terminal program here, but you can use another if you prefer. In order to connect via USB, you'll need to discover the device name to use. Review Chapter 7, "USB Serial," if you need help with this.

Tip   If your minicom is installed so that the default save directory requires root permission, you may want to use sudo minicom -s.

To set up minicom to use this, use the following:

```
$ minicom -s

  This brings up the following dialog:

          +-----[configuration]------+
          | Filenames and paths      |
          | File transfer protocols  |
          | Serial port setup        |<--- choose
          | Modem and dialing        |
          | Screen and keyboard      |
          | Save setup as dfl        |
          | Save setup as..          |
          | Exit                     |
          | Exit from Minicom        |
          +--------------------------+
```

Choose "Serial port setup" by using the cursor down key. Then, a setup dialog is shown, as follows:

```
+--------------------------------------------------------------+
| A -     Serial Device      : /dev/cu.usbserial-A100MX3L      |
| B - Lockfile Location      : /usr/local/Cellar/minicom/2.7/var |
| C -    Callin Program      :                                 |
| D -  Callout Program       :                                 |
| E -      Bps/Par/Bits      : 2400 8O1                        |
| F - Hardware Flow Control : No                               |
| G - Software Flow Control : No                               |
|                                                              |
|    Change which setting?                                     |
+--------------------------------------------------------------+
```

Type "A" and enter the device pathname for your USB device. The remaining settings are unimportant. Press Return to end that dialog, then save those settings by selecting "Save setup as..." I used the name of "usb." Saving your settings will save you time in the later chapters of this book.

Exit out of minicom by selecting "Exit from minicom." If you chose "Exit" instead and it didn't error back to the command line, you might need to use the Esc-X (or Control-A X) keystroke to exit. They must be typed quickly in succession to be recognized by minicom.

Running the Demo

With the minicom setup out of the way, you should be able to plug in your STM32's USB cable and start minicom (in that sequence). Use your saved settings name as the argument on the minicom command line (I used "usb" here):

```
$ minicom usb
```

Upon connecting through USB to your STM32, minicom should display the following:

```
Welcome to minicom 2.7

OPTIONS:
Compiled on Sep 17 2016, 05:53:15.
Port /dev/cu.usbmodemWGDEM1, 10:17:40

Press Meta-Z for help on special keys
```

Now we can communicate with the STM32. Press Return to prompt the demo code to show the following menu:

```
Winbond Flash Menu:
  0 ... Power down
  1 ... Power on
  a ... Set address
  d ... Dump page
  e ... Erase (Sector/Block/64K/Chip)
  i ... Manufacture/Device info
  h ... Ready to load Intel hex
  j ... JEDEC ID info
  r ... Read byte
  p ... Program byte(s)
  s ... Flash status
  u ... Read unique ID
  w ... Write Enable
  x ... Write protect

Address: 000000


:
```

Pressing any unrecognized command letter or pressing Return will cause this menu to redisplayed. The first thing to do is to see if we can read the flash device's status. Press "s" (or "S") to cause a status read:

```
: S
SR1 = 00 (write protected)
SR2 = 00
```

The demo program reports the W25Q25's SR1 as hex 00 and SR2 as 00. If you are seeing FF instead, then you may not be communicating over SPI correctly.

To Write Enable, press "W":

```
: W
SR1 = 02 (write enabled)
SR2 = 00
```

From this you can see that the flash device's SR1 now reads as hex 02, indicating that write is now enabled. With write enabled, you can perform a chip erase (press "E"):

```
: E

Erase what?
  s ... Erase 4K sector
  b ... Erase 32K block
  z ... Erase 64K block
  c ... Erase entire chip

anything else to cancel
: c
Erasing chip..
Chip erased!
```

Don't panic if the erase takes a few seconds. It's a dirty job, but somebody's got to do it. It's not a software hang, just the Winbond chip working hard.

Now, let's set the address and check if the chip is in fact erased. Type "A" and enter a zero, followed by Return:

```
: A
Address: 0
Address: 000000
```

Now, let's dump a page (256 bytes) by typing "D":

```
: D
000000 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
000010 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
000020 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
000030 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
000040 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
000050 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
000060 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
000070 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
000080 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
000090 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
0000A0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
0000B0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
0000C0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
0000D0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
0000E0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
0000F0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
Address: 000100
```

That seems to confirm an erasure, at least of page 0. Notice that the address has incremented by a page, so continued presses of "D" will allow the displaying of successive pages.

Now, let's write some bytes. Follow the session shown:

```
: W
SR1 = 02 (write enabled)
SR2 = 00

: A
Address: 0
Address: 000000

: P
$000000 AA BB CC DD EE
$000005 5 bytes written.
```

```
: D
000000 AA BB CC DD EE FF FF FF FF FF FF FF FF FF FF FF ................
000010 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
000020 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
```

In the session, we reset the address to zero after enabling writes and then typed "P" to program some bytes. In between each of the data bytes AA, BB, and so on, press Return. Pressing one extra Return will exit the program mode.

To verify that the data was written, page 0 was dumped. ASCII data can also be programmed by typing a quote character followed by the single character you want to enter. The following session illustrates this (page 0 assumed erased):

```
: P
$000000 'H 'e 'l 'l 'o '  'W 'o 'r 'l 'd '!
$00000C 12 bytes written.

: D
000000 48 65 6C 6C 6F 20 57 6F 72 6C 64 21 FF FF FF FF Hello World!....
000010 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
000020 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
```

It's a little tedious, but the text was entered successfully.

Now, let's test the nature of flash programming. Set the address to an erased location and program it as 0x7F. Here, we'll use address hex 10:

```
: P
$000010 7F
$000011 1 bytes written.

: D
000000 48 65 6C 6C 6F 20 57 6F 72 6C 64 21 FF FF FF FF Hello World!....
000010 7F FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
```

Check that the second line displays 7F at the left side. Now, program location hex 10, with the hex value BF:

```
: A
Address: 10
Address: 000010
```

```
: P
$000010 BF
$000011 1 bytes written.

: D
000000 48 65 6C 6C 6F 20 57 6F 72 6C 64 21 FF FF FF FF Hello World!....
000010 3F FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ?..............
```

Notice that the high bit (bit 7) of location 000010 is left unchanged. But because bit 6 (in 0xBF) was a zero, a new zero bit (bit 6) was programmed, leading to the resulting value of 0x3F.

Manufacturer ID

The identification of the flash chip can be tested with the "I" and "J" menu commands:

```
: I
Manufacturer $EF Device $15 (W25X32)

: J
Manufacturer $EF Type $40 Capacity $16 (W25X32)
```

Power Down

The Winbond chip can be powered down for current savings. Use menu options "0" and "1" to power off and on, respectively:

```
: 0

: 1
```

When on and reading status, the current draw on my unit was about 19.4 mA. When powered off, the current was reduced to 0.7 µA.

Summary

This has been a lengthy chapter, so give yourself a break. There are a couple of unexplored options in this demo program, like the unique ID and the Intel Hex upload. Do check out the unique ID feature. The Intel Hex upload feature will be used in the next chapter to program code overlays, so stay tuned for that. There are also a number of W25QXX features that were not explored, like its many protection features. To get the full scope of its capabilities, read the manufacturer datasheets. A simple Google search for "W25Q64 datasheet PDF" will find what you need.

Completion of this chapter means that now you are equipped with knowledge about the SPI protocol and how to apply it on the STM32 under FreeRTOS using libopencm3. That might seem like a lot of ducks to get in a row, and indeed it was. The next chapter will turn our attention to one practical use for external flash: code overlays.

Bibliography

1. "Serial Peripheral Interface Bus," Wikipedia, November 01, 2017. Accessed November 01, 2017. https://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus.

EXERCISES

1. How many data lines are used by SPI in bidirectional links? What are their signal names?

2. Where does the clock originate from?

3. What voltage levels are used for SPI signalling?

4. Why must a pull-up resistor be used for the STM32 /NSS line?

5. Why must a dummy value be sent in some SPI transactions?

Code Overlays

You don't hear much about code overlays today. With today's seemingly unlimited virtual memory in desktops and servers, applications often don't check for the risk of running out of memory. Yet in the early days of the mainframe's using core memory and the fledgling IBM PC, running out of memory was a frequent concern. Overlays were instrumental in doing more with less.

Overlays continue to have a role today in microcontrollers because of those products' own memory limits. Embedded products may begin with a selected microcontroller, only to discover later that the software won't fit. If this happens late in the product development cycle, a solution for using the existing MCU (Micro Controller Unit) must be found or software features must be dropped.

The designer may know that some sections of code are not needed often. A full-featured BASIC interpreter, for example, could swap in a code segment to renumber the program only when it is needed. The rest of the time, that code would remain unused and would not need to be resident.

There isn't much information available online about how to use GCC overlays.[1] There is plenty of discussion about load scripts, but specifics about the rest are usually left as an exercise for the reader. This chapter is dedicated to a *full* demonstration of a working example. This demo will swap overlays from the SPI flash chip into an SRAM overlay region, where the code will be executed. Given that these flash chips offer 4 MB or 8 MB of code storage, your imagination is the limit when it comes to larger applications on the STM32.

The Linker Challenge

In application development, your imagination leads you to write C code that is translated by the compiler into one or more object files (`*.o`). If the application is small enough, you link it into one final `*.elf` file, which is designed to fit the available flash memory

in the MCU. For the STM32, the st-flash utility needs a memory image file, so the following build step converts the .elf file to a binary image first:

```
$ arm-none-eabi-objcopy -Obinary main.elf main.bin
```

Then, the image file main.bin is uploaded to flash at address 0x8000000:

```
$ st-flash write main.bin 0x8000000
```

That is the *typical* link process, but how do you create overlays? Let's get started with the Winbond demo project. Go to the following subdirectory:

```
cd ~/stm32f103c8t6/rtos/winbond
```

Then, perform the following:

```
$ make clobber
$ make
```

This will force recompile that project, and at the end of it all the link step will look something like the following (the lines are broken up to fit the page for readability):

```
arm-none-eabi-gcc --static -nostartfiles -Tstm32f103c8t6.ld \
  -mthumb -mcpu=cortex-m3 -msoft-float -mfix-cortex-m3-ldrd \
  -Wl,-Map=main.map -Wl,--gc-sections main.o rtos/heap_4.o \
  rtos/list.o rtos/port.o rtos/queue.o rtos/tasks.o \
  rtos/opencm3.o -specs=nosys.specs -Wl,--start-group \
  -lc -lgcc -lnosys -Wl,--end-group \
  -L/Users/ve3wwg/stm32f103c8t6//rtos/libwwg -lwwg \
  -L/Users/ve3wwg/stm32f103c8t6/libopencm3/lib \
  -lopencm3_stm32f1 -o main.elf
```

This whole linking process is governed by the link script specified by the option -Tstm32f103c8t6.ld. When there is no -T option given on a Linux build command, for example, one will be assumed by default. But let's examine the file provided in the Winbond project.

MEMORY Section

The linker script contains a MEMORY section at the beginning that looks like the following:

```
MEMORY
{
        rom (rx) :  ORIGIN = 0x08000000, LENGTH = 64K
        ram (rwx) : ORIGIN = 0x20000000, LENGTH = 20K
}
```

This part of the load script declares two memory regions. These are regions that we are going to load code into (rom) or allocate space for (ram). If you are building large applications, I would advise you to change that rom size to 128K in the linker script and use the open-sourced st-link command to flash it using the --flash=128k option (doing a "make bigflash" will specify this option from the provided Makefile). As noted before, the STMF103C8T6 seems to support 128K despite its claim that only 64K exists.

After expanding rom to 128K, the MEMORY section should look like the following:

```
MEMORY
{
        rom (rx) :  ORIGIN = 0x08000000, LENGTH = 128K
        ram (rwx) : ORIGIN = 0x20000000, LENGTH = 20K
}
```

The *optional* attributes within brackets, like (rwx), describe the intended uses for the memory region (read, write, and execute). GCC documentation says that they are supported for backward compatibility with the AT&T linker but are otherwise only checked for validity.[2]

The ORIGIN = 0x20000000 parameter indicates where the block of ram memory physically resides. The LENGTH parameter is the size in bytes. Let's compare this notion of memory with the basic physical memory map of the MCU, shown in Figure 9-1.
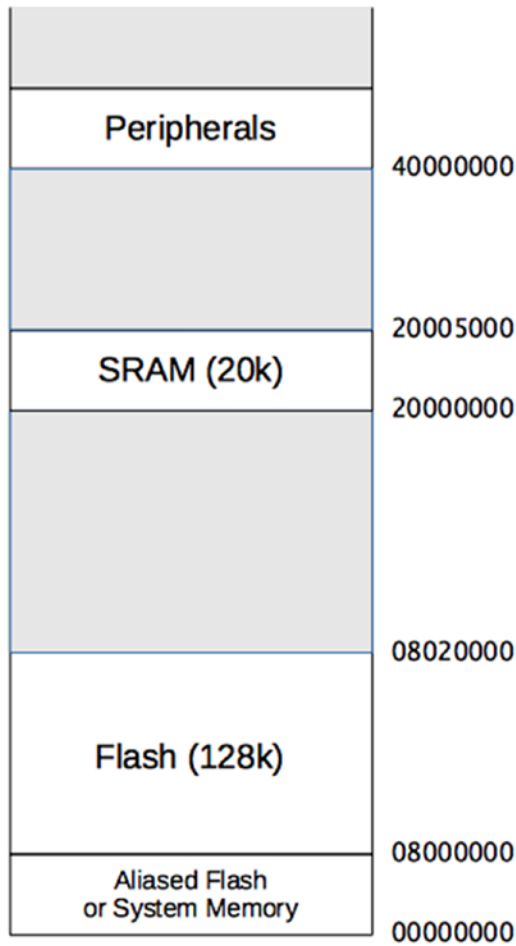
Figure 9-1. *Basic STM32F103C8T6 memory layout (addresses in hexadecimal)*

The memory that appears at the region starting at 0x00000000 depends upon the BOOT0 and BOOT1 switches. Normally, the setting BOOT0=0 is used, causing the flash memory to appear at location zero as well as at 0x08000000. This allows the MCU startup code to be the programmed flash memory.

At the higher address of 0x20000000 we find the static ram (SRAM). The size of this memory region is 20K for the STM32F103C8T6. Now, let's look at the remainder of the load script to see how it works.

Entry

The main driver of the load process is going to be the SECTIONS region of the file that we'll examine next, but there are two entries I'll discuss first. These are the ENTRY and EXTERN keywords:

```
ENTRY(reset_handler)
EXTERN (vector_table)
```

These entries do *not* appear in the MEMORY or SECTIONS areas of the load script but rather stand alone. The ENTRY keyword names the routine that is passed control at startup. The EXTERN keyword identifies a data area that will define the initial interrupt vector. With the environment being used, these will be supplied from the libopencm3 static library from a module named vector.o as follows:

```
~/stm32f103c8t6/libopencm3/lib/libopencm3_stm32f1.a
```

If you need to change the startup in any way, or are just curious, view the libopencm3 module here:

```
~/stm32f103c8t6/libopencm3/lib/cm3/vector.c
```

Because the symbol reset_handler is referenced by the ENTRY keyword, the vector.o module is loaded (unless you have supplied one of your own). This saves you from having to define all of the initialization required. When the reset handler has performed its setup, it will then call upon your main() program.

Sections

This is where things get interesting in the load script. In general terms, you'll find that this section appears like the following:

```
SECTIONS
{
    .text : {
        *(.vectors)     /* Vector table */
        *(.text*)       /* Program code */
        . = ALIGN(4);
        *(.rodata*)     /* Read-only data */
```

```
        . = ALIGN(4);
    } >rom
    ...etc...
}
```

I've trimmed some of the content so that you can focus on the essentials. From the snippet shown you can see that comments exist in the load script in C language form; for example, /* comment */. Don't let the remainder of the odd-looking syntax put you off. Let's break it down.

```
.text : {
    *(.vectors)      /* Vector table */
    *(.text*)        /* Program code */
    . = ALIGN(4);
    *(.rodata*)      /* Read-only data */
    . = ALIGN(4);
} >rom
```

A specific section begins with a name, which is .text in this example. Section names can be composed of almost any character, though odd characters or spaces must be quoted using ("). Otherwise, a symbol is expected to be surrounded by white space.

The section declared is named .text and is followed by a colon (:) and then a starting and ending curly brace. After the closing brace, we see >rom. This specifies that the input described between the curly braces will be placed into the MEMORY region named rom (remember that MEMORY section?)

What appears between the braces describes input and symbol calculations. Let's look at one input example first:

```
        *(.vectors)
```

What this means is that any input file (*) containing an object file section named .vectors is to be included in the output section being defined (in this case, .text).

Keep in mind that there are two kinds of sections involved:

1. Input object sections (like .vectors in the example)

2. Output sections (like .text in the example)

The initial asterisk names any file but could specify filenames instead. For example, the following two examples are possibilities:

```
*.o(.vectors)        /* any .o file having .vectors */
special.o(.vectors)  /* special.o having a .vectors section */
```

If we strip the example down to just the inputs, we would have the following:

```
.text : {
    *(.vectors)     /* Vector table */
    *(.text*)       /* Program code */
    *(.rodata*)     /* Read-only data */
} >rom
```

Boiled down from the preceding example, then, we are loading from *any* input file, from object file sections .vectors, .text, or .rodata. These will be loaded into the memory region named rom. Still with me?

Now, what about that other voodoo? The symbol dot (.) is used as the current location within the section. This practice undoubtedly comes from the assembler's use of the dot for the "location assignment counter." Within the link script, the dot symbol serves a similar purpose:

```
.text : {
    *(.vectors)     /* Vector table */
    *(.text*)       /* Program code */
    . = ALIGN(4);
    *(.rodata*)     /* Read-only data */
    . = ALIGN(4);
} >rom
```

The value of the dot at the middle (after .text) is the location at the end of the .text section (at that point in the link), but *rounded* up to the 4-byte word boundary (due to the use of special function ALIGN(4)). In this case, the current location is bumped up to the next aligned location. This is invoked a second time after the loading of input section .rodata (read-only data) so that if anything else is loaded into .text it will be aligned. Now, the mystery has been revealed!

Note that expressions like those involving dot are ended with a semicolon (`;`). Symbols can also be calculated in the same manner. For example, in the same script, and between section declarations, you'll find the following:

```
        ...
    } >rom

    . = ALIGN(4);
    _etext = .;
```

Because these last two lines are expressions appearing *outside* of the section definition (that is loading into `rom`), dot here will refer to the *last location referenced* (inside of `rom`). From this, we see that dot is aligned to the next word boundary and then assigned to a symbol `_etext`. Arithmetic is allowed in these expressions, if required. The symbol `_etext` in your program will then have the address of the first byte past the end of your read-only region in flash (`rom`).

PROVIDE

The `PROVIDE` keyword, used within a linker script, gives you the ability to define a symbol if it is needed (referenced). If the symbol isn't referenced, then its definition is withheld from the link to avoid symbol conflicts. The following will be found in your load script:

```
PROVIDE( _stack = ORIGIN(ram) + LENGTH(ram));
```

This statement says, provide symbol `_stack` if it is referenced. Calculate it as the *starting address* of memory region `ram` plus the *length* of the memory region. In other words, the starting address of a stack, which grows downward in the available SRAM region.

Relocation

As part of directing the linker, one issue that comes up is the need to have the linker put some stuff in flash memory, but to relocate references to that stuff as if it existed in SRAM. To look at this another way, we will be using some data in SRAM, but it will not live in SRAM until some startup code copies it there. Here is an example from your load script:

```
    .data : {
        _data = .;
```

```
    *(.data*)        /* Read-write initialized data */
    . = ALIGN(4);
    _edata = .;
 } >ram AT >rom
```

Here we define two symbols:

1.   _data (start of data)

2.   _edata (end of data)

Of particular interest is the last line:

```
 } >ram AT >rom
```

As you probably have guessed, this means that the symbols should be defined *as if they were* in ram (SRAM), *but* they will be written into the *flash* section (rom) instead. The initialization code within the module vector.o discussed earlier will *copy* this data from flash into the final SRAM location before main() is called.

This affects the relocation of any symbol references. If you had a static int constant, for example, that was not declared const, then it would be destined for SRAM. The address of that int will be set up by the linker to be somewhere in SRAM (address 0x20000000 to 0x20004FFF). However, the int value itself will be loaded into flash memory (somewhere between 0x08000000 and 0x0801FFFF). Startup initialization must copy it to SRAM to make it valid.

Keep this in mind as we turn our attention to overlays.

Defining Overlays

Now that you're armed and dangerous, let's get started with using the linker to define overlays. Change to the project directory overlay1:

```
$ cd ~/stm32f103c8t6/rtos/overlay1
```

In this project subdirectory, we have a modified version of the linker script stm32f103c8t6.ld that we'll be looking at.

The first thing of interest is that we've declared four memory regions instead of the usual two:

```
MEMORY
{
        rom (rx) :    ORIGIN = 0x08000000, LENGTH = 128K
        ram (rwx) :   ORIGIN = 0x20000000, LENGTH = 18K

        /* Overlay area in RAM */
        ovl (rwx) :   ORIGIN = 0x20004800, LENGTH = 2K

        /* Give external flash its own storage */
        xflash (r) :  ORIGIN = 0x00000000, LENGTH = 4M
}
```

The ram region has been shortened by 2K to leave room for the new SRAM overlay region named ovl. The memory addressed from 0x20004800 to 0x20004FFF is reserved to execute our overlay code.

The other new region, named xflash, is defined so that the linker can emit code that will reside in our external SPI flash. There will be more about this later.

The remainder of the linker script magic can be found later in the file as follows:

```
 OVERLAY : NOCROSSREFS {
     .fee {
         .overlay1_start = .;
         *(.ov_fee)              /* fee() */
         *(.ov_fee_data)         /* static data for fee() */
     }
     .fie { *(.ov_fie) }         /* fie() */
     .foo { *(.ov_foo) }         /* foo() */
     .fum { *(.ov_fum) }         /* fum() */
 } >ovl AT >xflash
 PROVIDE (overlay1 = .overlay1_start);
```

Let's now pick this apart. The OVERLAY keyword tells the linker to load all sections into the overlay section in an overlapping manner. In the example shown, the sections .fee, .fie, .foo, and .fum will all start at the same location. Given that >ovl puts this code into the overlay memory region, they will all have a starting address of 0x20004800. Of course, it is understood that not all of them can reside in that space at the same time.

The symbol `.overlay1_start` captures the starting address of the overlay region and is eventually passed into the `PROVIDE` statement so that symbol `overlay1` will contain the overlay starting address 0x20004800. This symbol can be used within the C program.

The keyword `NOCROSSREFS` provides another important linker feature. It would be unworkable for one overlay to call upon or reference another overlay in the same region. Only one overlay can reside in a region at one time. Calling function `fie()` from `fee()` would be disastrous. The `NOCROSSREFS` keyword instructs the linker to treat this scenario as an error.

Finally, note the following line:

```
 } >ovl AT >xflash
```

This directs the linker to relocate the code as if it runs at the overlay (`ovl`) address (in SRAM) but to place that overlay code into the memory region `xflash` instead. The `xflash` memory region will require a bit of special handling later on, but we need the linker to do this bit of trickery first.

An important concept here is that whatever goes into `xflash` is destined for the Winbond SPI flash device, starting at SPI flash address zero. This was established by the `ORIGIN` keyword in the following:

```
        xflash (r) :  ORIGIN = 0x00000000, LENGTH = 4M
```

Overlay Code

The section declared as `.fee` consists of two input sections, which come from the `.ov_fee` and `.ov_fee_data` sections. This provides an example of declaring code and data within the same overlay, presented in Listing 9-1.

Listing 9-1. The fee() Function Overlay Declaration

```
0027: int fee(int arg) __attribute__((noinline,section(".ov_fee")));
      ...
0115: int
0116: fee(int arg) {
0117:   static const char format[] // Placed in overlay
0118:       __attribute__((section(".ov_fee_data")))
0119:       = "***********\n"
0120:         "fee(0x%04X)\n"
```

```
0121:            "***********\n";
0122:
0123:    std_printf(format,arg);
0124:    return arg + 0x0001;
0125: }
```

To tell the compiler that the fee() function is to go to section .ov_fee (in the *object* file), we must use the GCC __attribute__ keyword (line 27). This attribute can *only* be specified in the function prototype.

The noinline keyword prevents GCC from inlining the code for fee(). This is especially important for our demo because the function is small enough to be inlined at the point of the call by GCC.

The second argument, section(".ov_fee"), names the section that our fee() function code should be written to in the main.o object file. The read-only data declared in lines 117 to 121 is specified to go into section .ov_fee_data. The compiler insists that this data section be different from the function code.

The remaining functions are simpler but apply the same idea. Listing 9-2 illustrates the fie() overlay function.

Listing 9-2. The fie() Function Overlay Code

```
0028: int fie(int arg) __attribute__((noinline,section(".ov_fie")));
      ...
0131: int
0132: fie(int arg) {
0133:
0134:    std_printf("fie(0x%04X)\n",arg);
0135:    return arg + 0x0010;
0136: }
```

Again, the overlay is named in the function prototype (line 28). The declaration of the function in lines 131 to 136 is per usual. Note that unlike fee(), the string constant used in line 134 will become part of the non-overlay code in the rom region here.

Overlay Stubs

Before the overlay code can be executed, the code from the SPI flash must be copied into the overlay area in SRAM. For this reason, each of the overlay functions uses a "stub function" and overlay manager, like the one shown in Listing 9-3.

Listing 9-3. Stub Function for fee()

```
0164: static int
0165: fee_stub(int arg) {
0166:   int (*feep)(int arg) = module_lookup(&__load_start_fee);
0167:
0168:   return feep(arg);
0169: }
```

Our fee() function takes an int argument and returns an int value. Consequently, the stub function must do the same. However, before we can call the overlay function, the function module_lookup() is invoked to see if it is *already* in the ovl (overlay) region and, if not, to copy it there now. Finally, we need to know its function address so that we can call it, which the module_lookup() function will return.

Overlay Manager

An overlay manager of some sort is usually required, *especially* when multiple overlay regions are used. Our demo program sets up an overlay table using an array of struct s_overlay:

```
0036: typedef struct {
0037:   short         regionx; // Overlay region index
0038:   void          *vma;    // Overlay's mapped address
0039:   char          *start;  // Load start address
0040:   char          *stop;   // Load stop address
0041:   unsigned long size;    // Size in bytes
0042:   void          *func;   // Function pointer
0043: } s_overlay;
```

For this demo, only one overlay region is used; therefore, regionx is always the index value zero. However, if you were to support three overlay regions, for example, this index could be a value of 0, 1, or 2. It is used to track which overlay is currently in the overlay region so that it is not always necessary to copy in the code.

The member vma is the overlay's *mapped* address (its SRAM location when executed). Members start and stop are the external flash addresses (from region xflash) that we need to load. Member size will have the calculated overlay size in bytes, while the final member func will contain the SRAM function pointer.

Are you still mulling over the values of start and stop right now? Give yourself points if you are. The question is how does the demo program locate the SPI flash code to load?

VMA and Load Addresses

The VMA (virtual memory address) and the load address for overlays are different. We have arranged for the overlay code and data to be written into the xflash memory area. Those *load* addresses will start from zero, since that is where the SPI flash addresses will begin. The VMAs for that code will be calculated for the overlay area in SRAM.

This is pointed out because we cannot use the VMAs for our overlay table as they map to the same region of SRAM. Some of the function pointers might even be the same. However, the load addresses (from SPI flash) will be unique. This permits us to use them as identifiers in our overlay table.

In the demo program a macro is used for programming convenience:

```
0048: #define OVERLAY(region,ov,sym) \
      { region, &ov, &__load_start_ ## sym, &__load_stop_ ## sym, 0, sym }
```

Because we are using only one overlay region, the region parameter will always be zero. But if you choose to add another, then you can supply the index as parameter 1.

The ov parameter refers to the overlay's starting address. The sym parameter allows us to specify the overlay function. Let's expand on this after we illustrate the demo program's table:

```
0056: // Overlay table:
0057: static s_overlay overlays[N_OVLY] = {
0058:   OVERLAY(0,overlay1,fee),
0059:   OVERLAY(0,overlay1,fie),
```

```
0060:    OVERLAY(0,overlay1,foo),
0061:    OVERLAY(0,overlay1,fum)
0062: };
```

In the demo table's contents, the symbol `overlay1` is referenced as the symbol describing the overlay's starting address in SRAM. The load script defines the start of that region as address 0x20004800 (for 2K bytes). Recall that the symbol was defined in the load script as follows:

```
PROVIDE (overlay1 = .overlay1_start);
```

Looking closer at one table entry,

```
0058:    OVERLAY(0,overlay1,fee),
```

we see that argument three is supplied as `fee`. The macro expands into the following line:

```
{ 0, &overlay1, &__load_start_fee, &__load_stop_fee, 0, fee }
```

Where do the symbols `__load_start_fee` and `__load_stop_fee` come from? These are automatically generated by the linker when the section `.fee` is processed. These two lines can be found in your `main.map` file that is written by the linker:

```
0x0000000000  PROVIDE (__load_start_fee, LOADADDR (.fee))
0x0000000045  PROVIDE (__load_stop_fee, (LOADADDR (.fee) + SIZEOF (.fee)))
```

From this we learn that the `.fee` section is loaded at address zero in the `xflash` (SPI flash) memory region and is 0x45 bytes (69 bytes) long.

Linker Symbols in Code

One thing that trips up new players when using linker symbols like `__load_start_fee`, for example, is that they try to use the *values* at those addresses rather than the *addresses* themselves. Let's clear this up with a code example:

```
extern long __load_start_fee;
```

Which is the correct usage to access the linker symbol `__load_start_fee`? Is it:

1. `__load_start_fee` (the *value*), or

2. `&__load_start_fee` (the *address*) ?

I've already given it away. Solution 2 is the correct answer, but why?

Solution 1 would imply that that the linker put 4 bytes of storage at the address of __load_start_fee, *containing* the symbol's value (which is an *address*). But the linker defines a symbol's value *as an address*, so *no storage is allocated*.

Returning to the overlay table that is used by the overlay manager, we see that the structure members of the first entry are populated as follows:

```
0036: typedef struct {
0037:    short          regionx;  // 0 (overlay index)
0038:    void           *vma;     // &overlay1
0039:    char           *start;   // &__load_start_fee
0040:    char           *stop;    // &__load_stop_fee
0041:    unsigned long  size;     // 0 (initially)
0042:    void           *func;    // A pointer inside SRAM
0043: } s_overlay;
```

This entry then defines the address of the SRAM overlay area in struct member vma using the linker-provided address &overlay1. Likewise, members start and stop also use *linker*-provided addresses. The size member will be calculated once at runtime. Finally, the member func is provided the value fee. What? What's going on with that?

Because the compiler knows that fee is the symbol of a *function* entry point of the function fee(), the simple reference to the symbol serves as the address. This linker-symbol mambo can be a little confusing.

Overlay Manager Function

Let's finally present the overlay function (Listing 9-4). The value that is passed in as the argument module is the overlay *load* address; for example, &__load_start_fee. This is the *address* that the linker placed the overlay code in, which will come from the SPI flash.

Listing 9-4. The Overlay Manager Function

```
0071: static void *
0072: module_lookup(void *module) {
0073:    unsigned regionx;          // Overlay region index
0074:    s_overlay *ovl = 0;        // Table struct ptr
0075:
```

```
0076:    std_printf("module_lookup(%p):\n",module);
0077:
0078:    for ( unsigned ux=0; ux<N_OVLY; ++ux ) {
0079:        if ( overlays[ux].start == module ) {
0080:            regionx = overlays[ux].regionx;
0081:            ovl = &overlays[ux];
0082:            break;
0083:        }
0084:    }
0085:
0086:    if ( !ovl )
0087:        return 0;                    // Not found
0088:
0089:    if ( !cur_overlay[regionx] || cur_overlay[regionx] != ovl ) {
0090:        if ( ovl->size == 0 )
0091:            ovl->size = (char *)ovl->stop - (char *)ovl->start;
0092:        cur_overlay[regionx] = ovl;
0093:
0094:        std_printf("Reading %u from SPI at 0x%04X into 0x%04X\n",
0095:            (unsigned)ovl->size,
0096:            (unsigned)ovl->start,
0097:            (unsigned)ovl->vma);
0098:
0099:        w25_read_data(SPI1,(unsigned)ovl->start,ovl->vma,ovl->size);
0100:
0101:        std_printf("Returned...\n");
0102:        std_printf("Read %u bytes: %02X %02X %02X...\n",
0103:            (unsigned)ovl->size,
0104:            ((uint8_t*)ovl->vma)[0],
0105:            ((uint8_t*)ovl->vma)[1],
0106:            ((uint8_t*)ovl->vma)[2]);
0107:    }
0108:    return ovl->func;
0109: }
```

Lines 78 to 84 perform a linear search of the table looking for a match on the module address (matching occurs in line 79). If a match is found, the *index* of the entry is saved in `regionx` (line 80). Then, the address of the overlay table entry is captured in line 81 in `ovl` before breaking out of the loop.

If the loop was exited without a match, 0 (null) is returned in line 87. This is fatal if used as a function call and indicates a bug in the application.

Line 89 checks to see if the overlay is valid and is already loaded or not. If the overlay must be read in, lines 90 to 107 are executed to make the overlay ready for use. If the overlay size is not yet known, it is calculated and saved in the table at lines 90 to 91. Line 92 tracks which overlay is currently loaded. Line 99 performs the SPI read from the flash device from the device's flash address `ovl->start` into the overlay SRAM memory at `ovl->vma` for `ovl->size` bytes.

With the overlay code loaded, the function pointer is returned in line 108.

Overlay Stubs

To ease the use of overlays, a *stub function* is normally used as a surrogate so that it can be called like a regular function. Listing 9-5 illustrates the stub function for the overlay `fee()`.

Listing 9-5. The fee() Stub Function

```
0164: static int
0165: fee_stub(int arg) {
0166:   int (*feep)(int arg) = module_lookup(&__load_start_fee);
0167:
0168:   return feep(arg);
0169: }
```

The stub function merely calls the overlay manager with the correct symbol (`&__load_start_fee` in this case). Once it has the function pointer captured in `feep`, it is safe to make the function call because the overlay manager can load the code when necessary. The function pointer `feep` allows the function to be invoked with the correct arguments and return the overlay's return value.

Demonstration

The demonstration program `main.c` (Listing 9-6) performs some initialization for SPI and for USB. Then, `task1` is launched to perform USB terminal I/O.

Listing 9-6. Initialization

```
0247: int
0248: main(void) {
0249:
0250:    rcc_clock_setup_in_hse_8mhz_out_72mhz(); // Use this for "blue
pill"
0251:    rcc_periph_clock_enable(RCC_GPIOC);
0252:    gpio_set_mode(GPIOC,GPIO_MODE_OUTPUT_2_MHZ,
             GPIO_CNF_OUTPUT_PUSHPULL,GPIO13);
0253:
0254:    usb_start(1);
0255:    std_set_device(mcu_usb);              // Use USB for std I/O
0256:
0257:     w25_spi_setup(SPI1,true,true,true,SPI_CR1_BAUDRATE_FPCLK_
DIV_256);
0258:
0259:    xTaskCreate(task1,"task1",100,NULL,configMAX_PRIORITIES-1,NULL);
0260:    vTaskStartScheduler();
0261:    for (;;);
0262:    return 0;
0263: }
```

To rebuild this project from scratch, perform:

```
$ make clobber
$ make
```

But don't flash your STM32 just yet.

Extracting Overlays

Before you can exercise your overlays, you have to get that overlay code loaded *onto* your W25Q32 flash device. Recall that we placed the overlay code in linker memory region xflash? Now we have to get that from the linker output and load it into the SPI device.

You may have noticed that the make command performed some extra steps in this project:

```
arm-none-eabi-gcc --static -nostartfiles -Tstm32f103c8t6.ld ... -o main.elf
for v in fee fie foo fum ; do \
    arm-none-eabi-objcopy -O ihex -j.$v main.elf $v.ov ; \
    cat $v.ov | sed '/^:04000005/d;/^:00000001/d' >>all.hex ; \
done
arm-none-eabi-objcopy -Obinary -R.fee -R.fie -R.foo -R.fum main.elf main.bin
```

After the normal link step (arm-none-eabi-gcc), you see some additional shell commands being issued as part of a for loop. For each of the overlay sections (fee, fie, foo, and fum) a pair of commands is issued, as follows:

```
    arm-none-eabi-objcopy -O ihex -j.$v main.elf $v.ov
    cat $v.ov | sed '/^:04000005/d;/^:00000001/d' >>all.hex
```

The first command extracts the named section in Intel hex format output (-O ihex). If variable v is the name fee, section .fee (-j.fee) is extracted to the file named fee.ov. The sed command that follows just strips out type 05 and 01 records from the hex file that we don't need and concatenates them all to the file all.hex.

The last step requires that we remove the overlay sections from main.elf so that the final image file doesn't include the overlays. If we left them in, then st-flash would try to upload that to the STM32 and fail.

```
arm-none-eabi-objcopy -Obinary -R.fee -R.fie -R.foo -R.fum main.elf main.bin
```

This command writes the image file main.bin (option -Obinary) and removes sections .fee, .fie, .foo, and .fum using the -R option. The main.bin is the image file that the st-flash command will use for the upload.

Tip   To make it easier to access from minicom, you may want to copy the file all.hex to your home directory or /tmp.

Upload Overlays to W25Q32

To upload the overlay code to the Winbond flash chip, use the project winbond to do it, from the project directory:

```
cd ~/stm32f103c8t6/rtos/winbond
```

Rebuild that project and flash it to your STM32:

```
$ make clobber
$ make
$ make flash
```

Before starting minicom, however, make sure that you have the following command installed on your system:

```
$ type ascii-xfr
ascii-xfr is /usr/local/bin/ascii-xfr
```

This is normally installed with minicom and may be installed in a different directory on your system. If not found, you'll need to fix that (maybe re-install minicom).

Then, disconnect the programmer and plug in the USB cable. Start up minicom:

```
$ minicom usb
```

With minicom running, check your upload settings next. Press Esc-O (or use Control-A O if necessary) quickly to bring up a menu, then select "File Transfer Protocols." If a menu didn't pop up, then try again. There cannot be much delay between typing the Escape/Control-A key and the letter O (oh).

Look for the protocol name "ascii," which is usually at the end of the list. Type the letter for the entry (letter I on my system), and press Return to enter the "Program" input area. Modify that entry to look as follows:

```
/usr/local/bin/ascii-xfr -n -e -s -l75
```

The most important option is the -l75 (lowercase el), which causes a 75 ms delay after each text line is sent. Without a reasonable delay, the uploads will fail. You probably should also set the other options as shown.

The remaining option flags are known to work:

```
Name U/D FullScr IO-Red. Multi
 Y   U    N        Y       N
```

Press Return to move through the list of input settings. Press Return one more time to pop back to the main menu, then select "Save setup as USB." You should now be able to use the minicom session to upload the all.hex file.

Once out of the menu, or in minicom initially, press Return to cause the program to present a menu:

```
Winbond Flash Menu:
  0 ... Power down
  1 ... Power on
  a ... Set address
  d ... Dump page
  e ... Erase (Sector/Block/64K/Chip)
  i ... Manufacture/Device info
  h ... Ready to load Intel hex
  j ... JEDEC ID info
  r ... Read byte
  p ... Program byte(s)
  s ... Flash status
  u ... Read unique ID
  w ... Write Enable
  x ... Write protect

Address: 000000


:
```

Check that your SPI flash is responding and erase it if necessary.

```
: W
SR1 = 02 (write enabled)
SR2 = 00

: E

Erase what?
  s ... Erase 4K sector
  b ... Erase 32K block
  z ... Erase 64K block
  c ... Erase entire chip
```

```
anything else to cancel
: s
sector erased, starting at 000000
Sector erased.

:
```

Here, our address is still zero, but if not set it to zero now:

```
: A
Address: 0
Address: 000000

:
```

Enable write again (erase disables it) and then prepare to upload the hex file:

```
: W
SR1 = 02 (write enabled)
SR2 = 00

: H

Ready for Intel Hex upload:
00000000 _
```

Now press Escape-S (or Control-A S) to pop up the Upload menu and choose "ascii":

```
        +-[Upload]--+
        | zmodem    |
        | ymodem    |
        | xmodem    |
        | kermit    |
        | ascii     |<-- Choose
        +-----------+
```

Another menu will pop up to allow you to choose a file to upload. I recommend just pressing Return and entering the file name (all.hex). I copy mine to the home directory so that I only need to type in "all.hex."

```
+-----------------------------------------+
|No file selected - enter filename:       |
|> all.hex                                |
+-----------------------------------------+
```

Upon pressing Return, an upload window pops up and sends the all.hex Intel hex code up to your STM32.

To check that it got there, you can dump the page, as follows:

```
: D
000000 10 B5 04 46 01 46 02 48 00 F0 06 F8 60 1C 10 BD ...F.F.H....`...
000010 20 48 00 20 00 00 00 00 5F F8 00 F0 FD 18 00 08  H. ...._.......
000020 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 0A 66 65 65 28 ***********.fee(
000030 30 78 25 30 34 58 29 0A 2A 2A 2A 2A 2A 2A 2A 2A 0x%04X).********
000040 2A 2A 2A 0A 00 10 B5 04 46 01 46 03 48 00 F0 06 ***.....F.F.H...
000050 F8 04 F1 10 00 10 BD 00 BF 30 31 00 08 5F F8 00 .........01.._..
000060 F0 FD 18 00 08 10 B5 04 46 01 46 03 48 00 F0 06 ........F.F.H...
000070 F8 04 F5 00 70 10 BD 00 BF 3D 31 00 08 5F F8 00 ....p....=1.._..
000080 F0 FD 18 00 08 10 B5 04 46 01 46 03 48 00 F0 06 ........F.F.H...
000090 F8 04 F5 40 50 10 BD 00 BF 4A 31 00 08 5F F8 00 ...@P....J1.._..
0000A0 F0 FD 18 00 08 FF FF FF FF FF FF FF FF FF FF FF ................
0000B0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
0000C0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
0000D0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
0000E0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
0000F0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
```

You should be able to see the text used by the fee() program's printf() string in the ASCII portion of the dump at right. You're now done with the flash memory upload!

Tip   Always exit minicom (Esc-X) prior to unplugging the USB cable. Otherwise, the USB driver can get hung or disabled.

Overlay Demo Continued

Now, exit minicom and unplug the USB cable, then return to the overlay1 project directory:

```
$ cd ~/stm32f103c8t6/rtos/overlay1
```

Flash the STM32 with the overlay code (main.bin):

```
$ make flash
```

Upon completion, unplug the programmer and plug in the USB cable. Enter minicom:

```
SPI SR1 = 00
Enter R when ready:
_
```

At this point, the demo program is waiting for your permission to try executing the overlays. Press "R" to try it:

```
OVERLAY TABLE:
[0] { regionx=0, vma=0x20004800, start=0x0, stop=0x45, \
      size=0, func=0x20004801 }
[1] { regionx=0, vma=0x20004800, start=0x45, stop=0x65, \
      size=0, func=0x20004801 }
[2] { regionx=0, vma=0x20004800, start=0x65, stop=0x85, \
      size=0, func=0x20004801 }
[3] { regionx=0, vma=0x20004800, start=0x85, stop=0xa5, \
      size=0, func=0x20004801 }
fang(0x0001)
module_lookup(0x0):
Reading 69 from SPI at 0x0000 into 0x20004800
Returned...
Read 69 bytes: 10 B5 04...
**********
fee(0x0001)
**********
module_lookup(0x45):
Reading 32 from SPI at 0x0045 into 0x20004800
Returned...
```

171

```
Read 32 bytes: 10 B5 04...
fie(0x0002)
module_lookup(0x65):
Reading 32 from SPI at 0x0065 into 0x20004800
Returned...
Read 32 bytes: 10 B5 04...
foo(0x0012)
module_lookup(0x85):
Reading 32 from SPI at 0x0085 into 0x20004800
Returned...
Read 32 bytes: 10 B5 04...
fum(0x0212)
calls(0xA) returned 0x3212

It worked!!


SPI SR1 = 00
Enter R when ready:
```

If your demo program gets as far as saying "It worked!!" and prompting you again for an "R," then your overlays worked. Notice that the sizes are zero initially in the dump of the overlay table. But if you type "R" again, you'll see that the size in bytes has been filled in:

```
OVERLAY TABLE:
[0] { regionx=0, vma=0x20004800, start=0x0, stop=0x45, \
      size=69, func=0x20004801 }
[1] { regionx=0, vma=0x20004800, start=0x45, stop=0x65, \
      size=32, func=0x20004801 }
[2] { regionx=0, vma=0x20004800, start=0x65, stop=0x85, \
      size=32, func=0x20004801 }
[3] { regionx=0, vma=0x20004800, start=0x85, stop=0xa5, \
      size=32, func=0x20004801 }
```

The size of .fee overlay is largest because we included some string text data with the code.

In the session output, the following can be disconcerting:

```
module_lookup(0x0):
Reading 69 from SPI at 0x0000 into 0x20004800
```

The first `&__load_start_fee` address used is SPI `flash` address 0x0 (not to be confused with a null pointer!). But that simply represents the first byte available in your SPI flash. The second line indicates that 69 bytes were loaded from flash at address 0x0000. We also see the reported overlay address of 0x20004800, which the code was loaded into for execution.

```
fie(0x0002)
module_lookup(0x65):
Reading 32 from SPI at 0x0065 into 0x20004800
```

From this we see that function `fie()` is called with an argument value of 2. It is located at address 0x65 in the SPI flash and loaded into the same overlay region at address 0x20004800.

Code Change Trap

Programmers are always looking for shortcuts, so I want to warn you about one trap that is easy to fall into. During this project's development, I made the assumption that I didn't need to re-upload the overlay file `all.hex` to the SPI flash because those routines didn't change. *However*, the location of the `std_printf()` routine they called *does* change in the non-overlay code.

The routines that your overlays call may move around as you change and recompile the code. When that happens, your overlay functions will crash when they call with the stale function addresses. Always update your overlay code even when the non-overlay code is changed.

Summary

This has been a technical chapter and was necessarily long. The benefit for you, however, is that you hold a complete recipe in your hands for implementing your own overlays. You are no longer confined to the STM32f103C8T6's flash limit of 128K. Spread your wings and fly!

EXERCISES

1. In the structure typedef'ed as `s_overlay`, why are members defined as character pointers rather than `long int`?

2. Why was the `xflash` memory region added to the linker script?

3. What is the purpose of the overlay stub function?

4. In the GNU declaration `__attribute__((noinline, section(".ov_fee")))`, what is the purpose of `noinline`? Why is it needed?

5. Where does the declaration `__attribute((section("...")))` belong?

Bibliography

1. "Overlay (programming)," Wikipedia, October 13, 2017. Accessed November 4, 2017. https://en.wikipedia.org/wiki/Overlay_(programming).

2. "Command Language," chapter in *Using LD, the GNU Linker,* 1d ed., by Steve Chamberlain. Accessed November 05, 2017. https://www.math.utah.edu/docs/info/ld_3.html#SEC13.

CHAPTER 10

Real-Time Clock (RTC)

Tracking time is often important in applications. For this reason, the STM32 platform provides a built-in real-time clock (RTC) peripheral. The datasheets for the RTC appear almost trivial to use, but there are wrinkles waiting for the unwary.

This chapter will examine how to set up interrupt service routines (ISR) for a recurring one-second interrupt event as well as the optional alarm feature. Armed with this information, there is no reason for your MCU applications to lack time information.

Demonstration Projects

The demonstration programs for this chapter come from the following two project directories:

```
$ cd ~/stm32f103c8t6/rtos/rtc
$ cd ~/stm32f103c8t6/rtos/rtc2
```

Initially the focus will be on the first project, where one ISR is implemented. The second example will apply the second alarm ISR, which will be examined near the end of the chapter.

RTC Using One Interrupt

The STM32F1 platform provides up to two interrupts for the RTC. The entry-point names when using libopencm3 are as follows:

```
#include <libopencm3/cm3/nvic.h>

void rtc_isr(void);
void rtc_alarm_isr(void);
```

Our first demonstration will use only the `rtc_isr()` routine because it is the simplest to set up and use. The serviced interruptible events are as follows:

1. One-second event (one-second timer tick)

2. Timer alarm event

3. Timer overflow event

The RTC can define one alarm such that an interrupt will be generated when the alarm expires at some time in the future. This can be used as a coarse-grained watchdog.

Even though the RTC counter is 32 bits in size, given enough time the counter will eventually overflow. Sometimes special time accounting must be performed at this point. The pending alarm may also need adjustment.

In our first demo, all of these optional events will be funneled through the `rtc_isr()` routine. This is the easiest approach.

RTC Configuration

The next sections will describe the configuration of the RTC clock using the libopencm3 library functions.

RTC Clock Source

The STM32F103C8T6 RTC can use one of three possible clock sources, as follows:

1. The LSE clock (32.768 kHz crystal oscillator), which continues to work even when the supply voltage is off, provided that the battery voltage V_{BAT} supply is maintained. The RTCCLK rate provided is 32.768 kHz. Setup choice is RCC_LSE.

2. The LSI clock (~40 kHz RC oscillator), but only while power is maintained. The RTCCLK rate provided is approximately 40 kHz. Setup choice is RCC_LSI.

3. The HSE clock (8 MHz crystal oscillator), but only while power is maintained. The RTCCLK rate provided is $\frac{8\,MHz}{128} = 62.5\,kHz$. Setup choice is RCC_HSE.

The clock source used by this chapter's project is the HSE clock, which is selected by the following libopencm3 function call:

```
rtc_awake_from_off(RCC_HSE);
```

Prescaler

Since we have chosen the RCC_HSE clock as the source for the RTC peripheral and are using the 8 MHz/128 = 62500 as the RTCCLK rate, we can now define the clock rate from the following formula:

$$f = \frac{62500}{divisor}Hz$$

We'll use the divisor of 62,500 in this chapter so that the frequency is 1 Hz. This provides a one-second tick time. You could, however, choose to use a divisor like 6,250 to produce a tick at tenth-second intervals, if required. Using libopencm3, we set the divisor as follows:

```
rtc_set_prescale_val(62500);
```

Starting Counter Value

Normally, the RTC counter would be started at zero, but there is no law that says you must. In the demo program, we're going to initialize it about 16 seconds before the counter overflows so that we can demonstrate the timer overflow interrupt.

The counter can be initialized with the following call:

```
rtc_set_counter_val(0xFFFFFFF0);
```

The RTC counter is 32 bits in size, so it overflows after counting to 0xFFFFFFFF. The preceding code initializes the counter to 16 counts prior to overflow.

RTC Flags

The RTC control register (RTC_CRL) contains three flags that we are interested in (using libopencm3 macro names):

1. RTC_SEC (tick)

2. RTC_ALR (alarm)

3. RTC_OW (overflow)

These flags can be tested and cleared, respectively, using the following libopencm3 calls:

```
rtc_check_flag(flag);
rtc_clear_flag(flag);
```

Interrupt and Setup

Listing 10-1 illustrates the steps necessary to initialize the RTC for one-second tick events and to receive interrupts.

Listing 10-1. The RTC Peripheral Setup for Interrupts

```
0166: static void
0167: rtc_setup(void) {
0168:
0169:    rcc_enable_rtc_clock();
0170:    rtc_interrupt_disable(RTC_SEC);
0171:    rtc_interrupt_disable(RTC_ALR);
0172:    rtc_interrupt_disable(RTC_OW);
0173:
0174:    // RCC_HSE, RCC_LSE, RCC_LSI
0175:    rtc_awake_from_off(RCC_HSE);
0176:    rtc_set_prescale_val(62500);
0177:    rtc_set_counter_val(0xFFFFFFF0);
0178:
0179:    nvic_enable_irq(NVIC_RTC_IRQ);
0180:
0181:    cm_disable_interrupts();
0182:    rtc_clear_flag(RTC_SEC);
0183:    rtc_clear_flag(RTC_ALR);
0184:    rtc_clear_flag(RTC_OW);
0185:    rtc_interrupt_enable(RTC_SEC);
0186:    rtc_interrupt_enable(RTC_ALR);
0187:    rtc_interrupt_enable(RTC_OW);
0188:    cm_enable_interrupts();
0189: }
```

Since the RTC peripheral is disabled after reset, it is enabled in line 169 so that it can be initialized. Lines 170 to 172 disable interrupts temporarily to prevent them while the peripheral is being set up.

Line 175 chooses the RTC clock source, and the clock rate is configured in line 176 to be once per second. Line 177 initializes the RTC counter to 16 seconds before overflow to demonstrate an overflow event without waiting a very long time.

Line 179 enables the interrupt controller for the `rtc_isr()` interrupt handler. All interrupts are temporarily suppressed in line 181 to allow the final interrupt setup to occur without generating interrupts. Lines 182 to 184 make sure that the RTC flags are cleared. Lines 185 to 187 enable the generation of interrupts when those flags are set. Last of all, interrupts are generally enabled once again at line 188.

At this point, the RTC peripheral is ready to generate interrupts.

Interrupt Service Routine

Listing 10-2 illustrates the code used to service the RTC interrupts. Don't let the length of the routine worry you, since there isn't really much going on there.

Listing 10-2. The RTC Interrupt Service Routine

```
0057: void
0058: rtc_isr(void) {
0059:    UBaseType_t intstatus;
0060:    BaseType_t woken = pdFALSE;
0061:
0062:    ++rtc_isr_count;
0063:    if ( rtc_check_flag(RTC_OW) ) {
0064:        // Timer overflowed:
0065:        ++rtc_overflow_count;
0066:        rtc_clear_flag(RTC_OW);
0067:        if ( !alarm ) // If no alarm pending, clear ALRF
0068:            rtc_clear_flag(RTC_ALR);
0069:    }
0070:
0071:    if ( rtc_check_flag(RTC_SEC) ) {
0072:        // RTC tick interrupt:
0073:        rtc_clear_flag(RTC_SEC);
```

```
0074:
0075:        // Increment time:
0076:        intstatus = taskENTER_CRITICAL_FROM_ISR();
0077:        if ( ++seconds >= 60 ) {
0078:            ++minutes;
0079:            seconds -= 60;
0080:        }
0081:        if ( minutes >= 60 ) {
0082:            ++hours;
0083:            minutes -= 60;
0084:        }
0085:        if ( hours >= 24 ) {
0086:            ++days;
0087:            hours -= 24;
0088:        }
0089:        taskEXIT_CRITICAL_FROM_ISR(intstatus);
0090:
0091:        // Wake task2 if we can:
0092:        vTaskNotifyGiveFromISR(h_task2,&woken);
0093:        portYIELD_FROM_ISR(woken);
0094:        return;
0095:    }
0096:
0097:    if ( rtc_check_flag(RTC_ALR) ) {
0098:        // Alarm interrupt:
0099:        ++rtc_alarm_count;
0100:        rtc_clear_flag(RTC_ALR);
0101:
0102:        // Wake task3 if we can:
0103:        vTaskNotifyGiveFromISR(h_task3,&woken);
0104:        portYIELD_FROM_ISR(woken);
0105:        return;
0106:    }
0107: }
```

An ISR counter named `rtc_isr_count` is incremented in line 62. This is only used to print the fact that the ISR was called in our demo code.

Lines 63 to 69 check to see if the counter overflowed, and if so clears the flag (`RTC_OW`). If there is no alarm pending, it also clears the alarm flag. It appears that the alarm flag is always set when an overflow happens, whether there was an alarm pending or not. As far as I can tell, this is one of those undocumented features.

Normally, the `rtc_isr()` routine is entered because of a timer tick. Lines 71 to 95 service the one-second tick. After clearing the interrupt flag (`RTC_SEC`), a critical section is begun in line 76. This is the FreeRTOS way to disable interrupts in an ISR until you finish with the critical section (line 89 ends the critical section). Here, it is critical that the time ripples up from seconds to minutes, hours, and days without being interrupted in the middle. Otherwise, a higher-priority interrupt could occur and discover that the current time is 12:05:60 or 13:60:45.

Line 92 is a notify check for task 2 (to be examined shortly). The notification (if `woken` is true) occurs in line 93. The idea is that task 2 will be blocked from executing until this notification arrives. Stay tuned for more about that.

Finally, lines 97 to 106 service the RTC alarm if the alarm flag is set. Aside from incrementing `rtc_alarm_count` for the demo print, it clears the alarm flag `RTC_ALR` in line 100. Lines 103 and 104 are designed to notify task 3, which will also be examined shortly.

Servicing Interrupts

The alert reader has probably noticed that the ISR routine didn't always service all three interrupt sources in one call. What happens if `RTC_SEC`, `RTC_OW`, and `RTC_ALR` are all set when the `rtc_isr()` routine is called but only `RTC_SEC` is cleared?

Any one of those flags may cause the interrupt to be raised. Since we enabled all three interrupt sources, the `rtc_isr()` routine will continue to be called until all flags are cleared.

Task Notification

FreeRTOS supports an efficient mechanism for allowing a task to block its own execution until another task or interrupt notifies it. Listing 10-3 illustrates the code used for task 2.

Listing 10-3. Task 2, Using Task Notify

```
0144: static void
0145: task2(void *args __attribute__((unused))) {
0146:
0147:   for (;;) {
0148:        // Block execution until notified
0149:        ulTaskNotifyTake(pdTRUE,portMAX_DELAY);
0150:
0151:        // Toggle LED
0152:        gpio_toggle(GPIOC,GPIO13);
0153:
0154:        mutex_lock();
0155:        std_printf("Time: %3u days %02u:%02u:%02u isr_count: %u,"
                              " alarms: %u, overflows: %u\n",
0156:            days,hours,minutes,seconds,
0157:            rtc_isr_count,rtc_alarm_count,rtc_overflow_count);
0158:        mutex_unlock();
0159:   }
0160: }
```

Like many tasks, it begins with an infinite loop in line 147. The first thing performed in that loop, however, is a call to ulTaskNotifyTake(), with a "wait forever" timeout (argument 2). Task 2 will grind to a halt there until it is notified. The only place it is notified is from the rtc_isr() routine in lines 92 and 93. Once the interrupt occurs, the function call in line 149 returns control and execution continues. This allows the print call in lines 155 to 157 to report the time.

When the demo runs, you will see that this notification occurs once per second as the rtc_isr() routine is called. This is a very convenient ISR to non-ISR routine synchronization. If you noticed the mutex_lock()/unlock calls, then just keep those in the back of your head for now.

Task 3 uses a similar mechanism for alarms, illustrated in Listing 10-4.

Listing 10-4. Task 3 and Its Alarm Notify

```
0126: static void
0127: task3(void *args __attribute__((unused))) {
0128:
```

```
0129:   for (;;) {
0130:       // Block execution until notified
0131:       ulTaskNotifyTake(pdTRUE,portMAX_DELAY);
0132:
0133:       mutex_lock();
0134:       std_printf("*** ALARM *** at %3u days %02u:%02u:%02u\n",
0135:           days,hours,minutes,seconds);
0136:       mutex_unlock();
0137:   }
0138: }
```

Line 131 blocks task 3's execution until it is notified by the rtc_isr() routine in lines 103 and 104 (Listing 10-2). In this manner, task 3 remains blocked until an alarm is sensed by the ISR.

Mutexes

Sometimes, mutexes (mutual-exclusion devices) are required to lock multiple tasks from competing for a shared resource. In this demo, we need to be able to format a complete line of text before allowing another task to do the same. The use of routines mutex_lock() and mutex_unlock() prevents competing tasks from printing in the middle of our own line of text. These routines use the FreeRTOS API and are shown in Listing 10-5.

Listing 10-5. Mutex Functions

```
0039: static void
0040: mutex_lock(void) {
0041:   xSemaphoreTake(h_mutex,portMAX_DELAY);
0042: }

0048: static void
0049: mutex_unlock(void) {
0050:   xSemaphoreGive(h_mutex);
0051: }
```

The handle to the mutex is created in the main program with the following call:

```
0259:   h_mutex = xSemaphoreCreateMutex();
```

If you're new to mutexes, the following happens when you lock (take) a mutex:

1. If the mutex is already locked, the calling task blocks (waits) until it becomes free.

2. When the mutex is free, an attempt will be made to lock it. If another competing task grabbed the lock first, return to step 1 to block until the mutex becomes free again.

3. Otherwise, the lock is now *owned* by the caller, until the mutex is unlocked.

Once the task using the mutex is done with it, it can release the mutex. The act of unlocking the mutex may allow another blocked task to continue. Otherwise, if no other task is waiting on the mutex, the mutex is simply unlocked.

Demonstration

The main demonstration program presented in this chapter can be run using a USB-based TTL UART or directly over a USB cable. The UART choice is perhaps the best since the USB link sometimes introduces delays that mask the timing of the print statements. However, both work equally well once things get going, and the USB cable is usually more convenient.

The following statement defines which approach you want to use (in file named main.c):

```
0020: #define USE_USB    0     // Set to 1 for USB
```

It defaults to UART use. If you set it to 1, the USB device will be used instead. The setup for UART or USB occurs in the main() function of main.c (Listing 10-6).

Listing 10-6. Main Program for Initializing UART or USB

```
0251: int
0252: main(void) {
0253:
0254:   rcc_clock_setup_in_hse_8mhz_out_72mhz(); // Use this for "blue pill"
0255:
0256:   rcc_periph_clock_enable(RCC_GPIOC);
0257:   gpio_set_mode(GPIOC,GPIO_MODE_OUTPUT_50_MHZ,
          GPIO_CNF_OUTPUT_PUSHPULL,GPIO13);
```

```
0258:
0259:    h_mutex = xSemaphoreCreateMutex();
0260:    xTaskCreate(task1,"task1",350,NULL,1,NULL);
0261:    xTaskCreate(task2,"task2",400,NULL,3,&h_task2);
0262:    xTaskCreate(task3,"task3",400,NULL,3,&h_task3);
0263:
0264:    gpio_clear(GPIOC,GPIO13);
0265:
0266: #if USE_USB
0267:    usb_start(1,1);
0268:    std_set_device(mcu_usb); // Use USB for std I/O
0269: #else
0270:    rcc_periph_clock_enable(RCC_GPIOA); // TX=A9,RX=A10,CTS=A11,RTS=A12
0271:    rcc_periph_clock_enable(RCC_USART1);
0272:
0273:    gpio_set_mode(GPIOA,GPIO_MODE_OUTPUT_50_MHZ,
0274:        GPIO_CNF_OUTPUT_ALTFN_PUSHPULL,GPIO9|GPIO11);
0275:    gpio_set_mode(GPIOA,GPIO_MODE_INPUT,
0276:        GPIO_CNF_INPUT_FLOAT,GPIO10|GPIO12);
0277:    open_uart(1,115200,"8N1","rw",1,1);  // RTS/CTS flow control
0278:    // open_uart(1,9600,"8N1","rw",0,0); // UART1 9600 with no f.control
0279:    std_set_device(mcu_uart1);        // Use UART1 for std I/O
0280: #endif
0281:
0282:    vTaskStartScheduler();
0283:    for (;;);
0284:
0285:    return 0;
0286: }
```

The device control is initialized in lines 267 and 268 for USB and lines 270 to 279 for UART. For the UART, the baud rate 115,200 is used by default in line 277. This works well if you use hardware flow control. If for some reason your TTL serial device doesn't support hardware flow control, then comment out line 277 and uncomment line 278 instead. At the lower baud rate of 9,600 you should be able to operate safely without flow control.

Lines 268 or 279 determine whether std_printf() et al. are directed to the USB device or to the UART device.

As we noted earlier, we used a mutex to guarantee that complete text lines would be printed to the console (USB or UART). This FreeRTOS mutex is created in line 259.

Task 1 (line 260) will be our main "console," allowing us to type characters to affect the operation of the demo. Task 2 (line 261) will toggle the LED (PC13) for each RTC tick, as well as print the current time since startup (see earlier Listing 10-3). Finally, task 3 will print an alarm notice when the alarm has been triggered (see earlier Listing 10-4).

Listing 10-7 illustrates the console task (task 1).

Listing 10-7. The "Console Task," Task 1

```
0220: static void
0221: task1(void *args __attribute__((unused))) {
0222:    char ch;
0223:
0224:    wait_terminal();
0225:    std_printf("Started!\n\n");
0226:
0227:    rtc_setup();     // Start RTC interrupts
0228:    taskYIELD();
0229:
0230:    for (;;) {
0231:        mutex_lock();
0232:        std_printf("\nPress 'A' to set 10 second alarm,\n"
0233:            "else any key to read time.\n\n");
0234:        mutex_unlock();
0235:
0236:        ch = std_getc();
0237:
0238:        if ( ch == 'a' || ch == 'A' ) {
0239:            mutex_lock();
0240:            std_printf("\nAlarm configured for 10 seconds"
0241:                            " from now.\n");
0241:            mutex_unlock();
```

```
0242:               set_alarm(10u);
0243:          }
0244:     }
0245: }
```

Task 1 is there to allow synchronization with the user who is connecting with the USB or UART link, using minicom. Line 224 calls `wait_terminal()`, which prompts the user to press any key, after which the function returns. Then, the RTC clock is initialized at line 227 and an initial `taskYIELD()` call is made. This helps to get everything prior to entering the task 1 loop.

The main loop of task 1 simply issues a printed message to press "A" to set a ten-second alarm. Any other key will just cause the console loop to repeat. Because the RTC timer is interrupt driven, the `rtc_isr()` method is called each second, or when overflow or alarm occurs. This in turn notifies task 2 or task 3 as previously discussed.

UART1 Connections

When you use the TTL UART, the connections are made according to Table 10-1. In this case, we can power the STM32 from the TTL UART device. If you are not powering the STM32 from the TTL UART, then omit the +5V connection.

Table 10-1. *UART Wiring to STM32F103C8T6*

GPIO	UART1	TTL UART	Description
A9 (out)	TX	RX	STM32 sends to TTL UART
A10 (in)	RX	TX	STM32 receives from TTL UART
A11 (out)	CTS	RTS	STM32 clear to send
A12 (in)	RTS	CTS	STM32 request to send
+5V		+5V	Powers STM32 from USB TTL UART (otherwise, omit this connection when STM32 is powered by another source).
Gnd		Gnd	Ground

> **Note**   UART1 was chosen because it uses 5-volt-tolerant GPIOs.

From a wiring and terminal-emulator standpoint, the USB cable is a much simpler option.

Running the Demo

Depending upon how you configured the main.c program to run, you will be using the USB cable or the TTL UART. USB is the simplest—just plug in and go. If you're using the UART, then you need to configure your terminal program (minicom) to match the communication parameters: baud rate, 8 data bits, no parity, and one stop bit.

Upon starting your terminal program (I'm using minicom), you should see a prompt to press any key:

```
Welcome to minicom 2.7

OPTIONS:
Compiled on Sep 17 2016, 05:53:15.
Port /dev/cu.usbserial-A703CYQ5, 19:13:38

Press Meta-Z for help on special keys


Press any key to start...
Press any key to start...
Press any key to start...
```

Press any key to get things rolling (I used the Return key):

```
Press any key to start...
Started!

Press 'A' to set 10 second alarm,
else any key to read time.

Time:   0 days 00:00:01 isr_count: 1, alarms: 0, overflows: 0
Time:   0 days 00:00:02 isr_count: 2, alarms: 0, overflows: 0
Time:   0 days 00:00:03 isr_count: 3, alarms: 0, overflows: 0
...
```

```
Time:    0 days 00:00:20 isr_count: 20, alarms: 0, overflows: 0
Time:    0 days 00:00:21 isr_count: 21, alarms: 0, overflows: 0
Time:    0 days 00:00:22 isr_count: 22, alarms: 0, overflows: 0
Time:    0 days 00:00:23 isr_count: 23, alarms: 0, overflows: 0
Time:    0 days 00:00:24 isr_count: 24, alarms: 0, overflows: 1
Time:    0 days 00:00:25 isr_count: 25, alarms: 0, overflows: 1
Time:    0 days 00:00:26 isr_count: 26, alarms: 0, overflows: 1
Time:    0 days 00:00:27 isr_count: 27, alarms: 0, overflows: 1
```

Once started, the rtc_isr() is enabled and the evidence of one-second interrupts is realized in the printed messages. The count value isr_count indicates how often the interrupt has invoked the ISR routine rtc_isr(). Notice that the overflows count increases when the isr_count is 24. This indicates that the RTC counter has overflowed and is now restarting from zero.

The elapsed time in days, hours, minutes, and seconds is shown in each message. These values were calculated in the critical section within the rtc_isr() routine.

To create an alarm, press "A." This will start an alarm that will expire in ten seconds. The following session begins the alarm near time 00:07:40, and the alarm message appears at 00:07:50 as expected:

```
Time:    0 days 00:07:38 isr_count: 458, alarms: 0, overflows: 1
Time:    0 days 00:07:39 isr_count: 459, alarms: 0, overflows: 1

Alarm configured for 10 seconds from now.

Press 'A' to set 10 second alarm,
else any key to read time.

Time:    0 days 00:07:40 isr_count: 460, alarms: 0, overflows: 1
Time:    0 days 00:07:41 isr_count: 461, alarms: 0, overflows: 1
...
Time:    0 days 00:07:48 isr_count: 468, alarms: 0, overflows: 1
Time:    0 days 00:07:49 isr_count: 469, alarms: 0, overflows: 1
*** ALARM *** at    0 days 00:07:50
Time:    0 days 00:07:50 isr_count: 471, alarms: 1, overflows: 1
Time:    0 days 00:07:51 isr_count: 472, alarms: 1, overflows: 1
```

Listing 10-8 shows the snippet of code that starts the ten-second alarm. It uses the libopencm3 routine set_alarm() in line 242. The function call sets up a register to trigger an alarm ten seconds from the present time.

Listing 10-8. The Alarm-Triggering Code

```
0236:        ch = std_getc();
0237:
0238:        if ( ch == 'a' || ch == 'A' ) {
0239:            mutex_lock();
0240:            std_printf("\nAlarm configured for 10 seconds"
                                    " from now.\n");
0241:            mutex_unlock();
0242:            set_alarm(10u);
0243:        }
```

rtc_alarm_isr()

If you've read any of the STM32F103C8T8 datasheet information, you're probably aware that there is a second possible ISR entry point. The datasheet is rather cryptic about this, unless you know what they mean by the "RTC global interrupt" and "EXTI Line 17" references. The entry point rtc_isr() routine *is* the "RTC global interrupt," while "EXTI Line 17" means something else.

The EXTI controller refers to the external interrupt/event controller. The purpose of this controller is to allow GPIO input lines to trigger an interrupt on a signal rise or fall event. So, I think that you'd be excused if you asked "What's *external* about RTC?" The demo code implementing the rtc_alarm_isr() interrupt is found in the following directory:

```
$ cd ~/stm32f103c8t6/rtos/rtc2
```

EXTI Controller

As previously mentioned, the EXTI controller allows GPIO input lines to trigger an interrupt if their input level rises or falls, depending upon the configuration (or both rise and fall). All GPIO 0's map to interrupt EXT0. For the STM32F103C8T6, this means GPIO

ports PA0, PB0, and PC0. Other STM32 devices with additional GPIO ports might also have PD0, PE0, etc. Likewise, the EXT15 interrupt is raised by GPIO ports PA15, PB15, PC15, etc.

This defines EXT0 through EXT15 as interrupt sources. But in addition to these, there are up to four more:

- EXT16 – PVD output (programmable voltage detector)

- EXT17 – RTC alarm event

- EXT18 – USB wakeup event

- EXT19 – Ethernet wakeup event (not on F103C8T6)

These are internal events that can also create interrupts. Of immediate interest is the RTC alarm event (EXT17).

Configuring EXT17

To get interrupts on rtc_alarm_isr(), we must configure event EXTI17 to raise an interrupt. This requires the following libopencm3 steps:

1. #include <libopencm3/stm32/exti.h>

2. exti_set_trigger(EXTI17,EXTI_TRIGGER_RISING);

3. exti_enable_request(EXTI17);

4. nvic_enable_irq(NVIC_RTC_ALARM_IRQ);

Step one is the additional libopencm3 include file required. Step two indicates what event we want as a trigger, and this configures the rising edge of the alarm event. Step three enables the EXTI17 interrupt in the peripheral. Finally, step four enables the interrupt controller to process the RTC_ALARM_IRQ event.

Because the alarm handling has been separated out from the rtc_isr() handler, the new rtc_alarm_isr() looks rather simple in Listing 10-9.

Listing 10-9. The rtc_alarm_isr() Routine

```
0098: void
0099: rtc_alarm_isr(void) {
0100:   BaseType_t woken = pdFALSE;
0101:
```

191

```
0102:    ++rtc_alarm_count;
0103:    exti_reset_request(EXTI17);
0104:    rtc_clear_flag(RTC_ALR);
0105:
0106:    vTaskNotifyGiveFromISR(h_task3,&woken);
0107:    portYIELD_FROM_ISR(woken);
0108: }
```

The handler is almost identical to the alarm-event handling presented earlier, but there is one more step that must be observed, as follows:

```
0103:    exti_reset_request(EXTI17);
```

This libopencm3 call is necessary to reset the EXTI17 interrupt in addition to usual clearing the flag RTC_ALR in line 104.

This additional setup for EXTI17 is not particularly burdensome but can be tricky to get working from the datasheets. Now that you've seen the secret sauce, this should be a no brainer.

Run the RTC2 demo the same way as RTC. The only difference between the two is the interrupt handling.

Summary

This chapter has explored the configuration and use of the real-time clock. From this presentation, it is clear that the RTC is not a complicated peripheral within the STM32. The utility of having a solid and accurate time should not be underappreciated, however.

Despite the simplicity, there are areas that require careful consideration, like correct handling of timer overflows. This becomes even more critical as higher-resolution units are used, like $\frac{1}{100}th$ second. When the alarm feature is used, the RTC counter overflow event may also require special handling after an overflow.

Knowing how to set up EXTI17 also permits you to set up GPIO signal-change interrupts. The procedure is the same, except that you specify EXTIn for GPIOn.

EXERCISES

1. What are the three possible interrupt events from the RTC?

2. What is the purpose of the calls `taskENTER_CRITICAL_FROM_ISR` and `taskEXIT_CRITICAL_FROM_ISR`?

3. How many bits wide is the RTC counter?

4. Which clock source continues when the STM32 is powered down?

5. Which is the most accurate clock source?

I2C

The I2C bus is a convenient hardware system mainly because it requires only two wires for communication. The bus is also known by other names, such as the IIC (inter-integrated circuit) or TWI (two-wire interface). Phillips Semiconductor developed the I2C bus, which Intel later extended with the SMBus protocol. These are largely interchangeable, but I will focus on I2C in this chapter.

With the utility of the I2C bus, it is no surprise that the STM32 platform includes a hardware peripheral for it. This chapter will explore how to utilize the peripheral in concert with the PCF8574 GPIO extender device attached to the bus.

The I2C Bus

One of the hallmarks of I2C as a serial communications bus is that it requires only two wires. The power supply and ground connections are not included in this count. The two communication lines involved are the following:

- System clock (usually labeled *SCL*)

- System data (usually labeled *SDA*)

Each of these lines rests at a high voltage level (usually at 5 or 3.3 volts). Any device on the bus can generate a data signal by pulling the line down low (zero volts). This works well for open-collector (bipolar) or open-drain (FET) transistors. When the transistor is *active*, they act like a switch shorting the bus line to ground. When the bus is *idle*, a pullup resistor pulls the voltage of the line high. For this reason, both I2C lines operate with at least one pullup resistor.

Master and Slave

With every device on the bus able to pull the lines low, there must be some sort of protocol to keep things organized. Otherwise, the bus would have multiple conversations going on with no receivers making sense of the garbled messages. For this reason, the I2C protocol often uses *one* master and many slave devices. In more-complex systems, it is possible to have more than one master controller, which is out of scope for this chapter.

The master device always starts the conversation and drives the clock signal. The exception to the SCL line's being driven by the master is that slaves can sometimes stretch the clock to buy extra time (when it is supported by the master). Clock stretching occurs when the slave device continues to hold the SCL line low after the master has released it.

Slave devices only respond when spoken to. Each slave has a unique 7-bit device address so that it knows when a bus message has been sent to it. This is one area where I2C differs from the SPI bus. Each I2C device is addressed by an address, while SPI devices are selected by a chip-select line.

Start and Stop

The I2C is idle when both the SDA and SCL lines are pulled high. In this case, no device—master or slave—is pulling the bus lines low.

The start of an I2C transaction is indicated by the following events:

1.   The SCL line remains high.

2.   The SDA line is pulled down.

Step two usually happens within a clock cycle, although it need not be precisely so. When the bus is idle, it is enough to see the SDA line going low while the SCL remains high. Figure 11-1 illustrates the start, stop, and repeated start I2C signals.
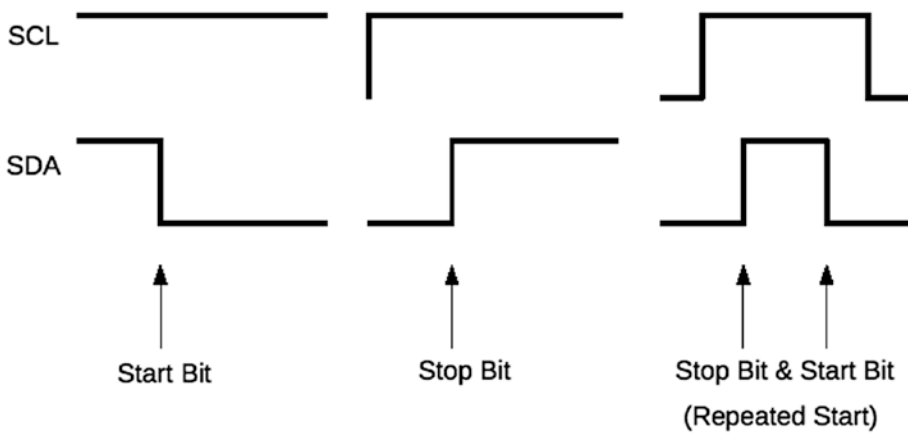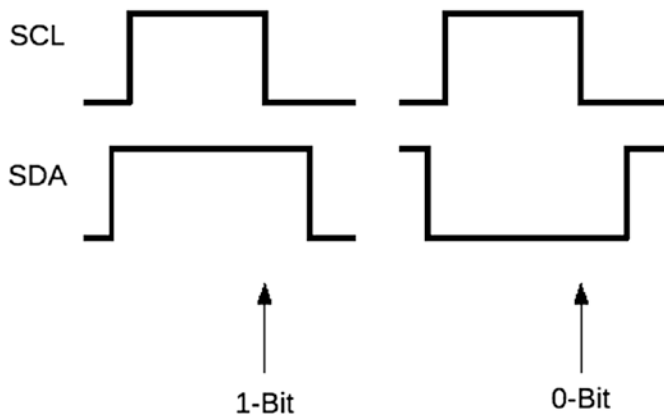
Figure 11-1. *I2C start, stop, and repeated start bit signals*

The repeated start is both an optimization of the stop and start and a way to hold the bus while continuing with a longer I2C transaction. Later, we'll discuss this further.

Data Bits

Data bits are transmitted in concert with a high-to-low transition in the clock (SCL) signal. Figure 11-2 illustrates.


Figure 11-2. *I2C data bit signals*

The sampling of the SDA bus line occurs where the arrows are shown. The high or low state of the SDA line is read at the point where the SCL line is pulled low (by the master).

I2C Address

Before we look at the whole bus transaction, let's describe the address byte, which is used to identify the slave device that is expected to respond (Figure 11-3). In addition to the address, a read/write bit indicates the intention to read or write from/to the slave device.
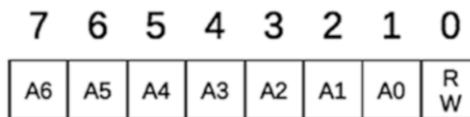
Figure 11-3. *I2C 7-bit address format*

The 7 bits of address are shifted up 1 bit in the address byte, while the read/write bit is rightmost. The read/write bit is defined as follows:

- 1-bit indicates that a read operation from the slave device will follow

- 0-bit indicates that a write operation to the slave device will follow

The address and read/write bit always follow a start or repeated start bit on the I2C bus. The start bit requires the I2C controller to check that the bus is not in use by another master; it does this by using a bus arbitration procedure (when multi-master is supported). But once bus access is won, the bus is owned by the controller until it is released with a stop bit.

The repeated start allows the current transaction to continue without further bus arbitration. Since an address and read/write bit must follow, this allows multiple slaves to be serviced with one transaction. Alternatively, the same slave may be addressed but be accessed with a different read/write mode.
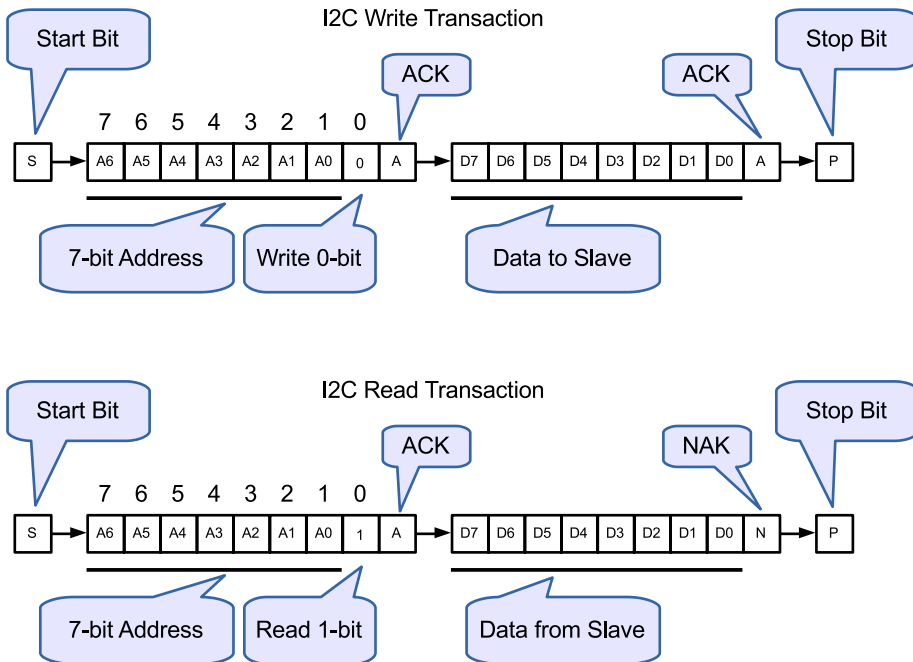

Tip   Sometimes people report that slave addresses shifted up by one bit as they were sent. This has the effect of multiplying the address by two. The address 0x42 when shifted right is actually 0x21. Watch out for this in documentation.

I2C Transactions

Figure 11-4 illustrates a write and a read transaction (where no repeated start is used).

Figure 11-4. *I2C write transaction and a read transaction*

The upper portion of Figure 11-4 illustrates a simple write transaction. The basic train of events for the illustrated write are as follows:

1. The I2C controller gains control of the bus and emits a start bit.

2. The master (controller) writes out seven address bits followed by a 0-bit, indicating that this will be a write transaction.

3. The slave device acknowledges the request and pulls the data line low during the ACK (acknowlege) bit. If no slave responds, the data line will float high and cause a NAK (negative acknowlege) to be received by the controller instead.

4. Because this is a write transaction, the data byte is written out.

5. The slave device acknowledges the receipt of the data byte by pulling down the data line during the ACK bit time.

6. The master is not sending any more data, so it writes a stop bit and releases the I2C bus.

The read request is similar:

1. The I2C controller gains control of the bus and emits a start bit.

2. The master (controller) writes out seven address bits followed by a 1-bit, indicating that this will be a read transaction.

3. The slave device acknowledges the request and pulls the data line low during the ACK bit. If no slave responds, the data line will float high and cause a NAK to be received by the controller instead.

4. Because this is a read transaction, the master continues to write out clock bits to allow the slave device to synchronize its data response back to the master.

5. With each clock pulse, the slave device writes out the eight data bytes to the master controller.

6. During the ACK time, the master controller normally sends a NAK when no more bytes are to be read.

7. The controller sends a stop bit, which always ends the transaction with the slave (regardless of the last ACK/NAK sent).

PCF8574 GPIO Extender

To exercise the I2C bus in this chapter, we'll be using the PCF8574 GPIO extender chip (Figure 11-5). This is a great chip for adding additional GPIO lines, provided that you don't need high speed (the demo operates the I2C bus at 100 kHz).
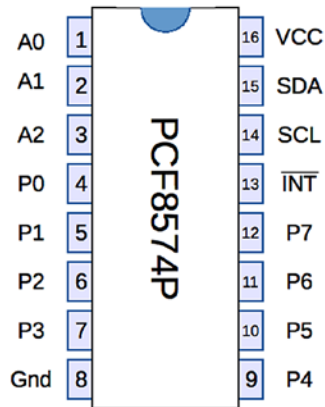
Figure 11-5. *Pinout of the PCF8574P*

The +3.3 volts power is applied to pin 16, while the grounded connection is pin 8. Pins A0 through A2 are used to select the chip's slave address (Table 11-1). Pins P0 through P7 are the GPIO databit lines, which can be input or output. Pin 14 connects to the clock line (SCL), while pin 15 connects to the data line (SDA). Pin 13 can be used for notification.

Table 11-1. *PCF8574 Address Configuration*

A0	A1	A2	PCF8574 Address	PCF8574A Address
0	0	0	0x20	0x38
0	0	1	0x21	0x39
0	1	0	0x22	0x3A
0	1	1	0x23	0x3B
1	0	0	0x24	0x3C
1	0	1	0x25	0x3D
1	1	0	0x26	0x3E
1	1	1	0x27	0x3F

Address lines A0 through A2 are programmed as zeros when grounded and as 1-bits when connected to V_{cc} (+3.3 volts in this demo). If you have the PCF8575A chip, then the address should be taken from the right column. The earlier PCF8574 chip uses hexadecimal addresses in the left column of the table.

I2C Circuit

Figure 11-6 illustrates three PCF8574P devices attached to the STM32 through the I2C bus.
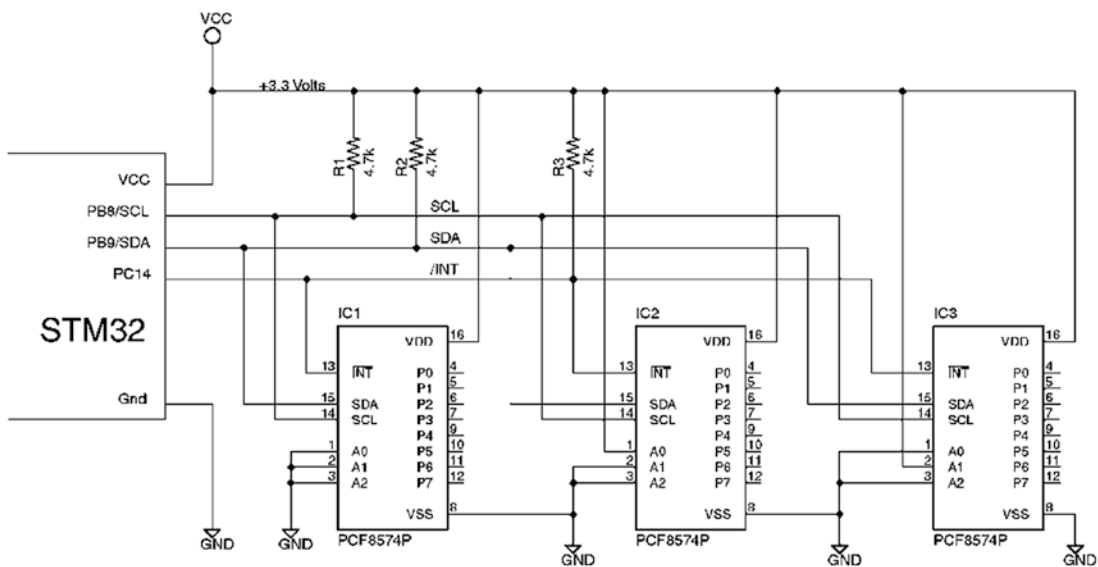
Figure 11-6. *STM32 attached to three PCF8574P slave devices using the I2C bus*

The schematic looks a little busy, but it's not that bad. Notice that the I2C bus consists only of a pair of lines, SCL and SDA, originating from the STM32. These two lines are pulled high by resistors R1 and R2, respectively. Each slave device is also connected to these bus lines, allowing each of these to respond when it recognizes its slave address.

Notice how IC1, IC2, and IC3 each have a different wiring for the address pins A0, A1, and A2. This configures each device to respond to a different slave address (review Table 11-1). These addresses must be unique.

The PCF8574 $\overline{INT}$ Line

The remaining connections in Figure 11-6 are power and the $\overline{INT}$ line. The $\overline{INT}$ line is an *optional* bus component that has nothing to do with the I2C bus itself. You might not even attach all PCF8574P devices to the $\overline{INT}$ line if they never get used for GPIO *input*.

The $\overline{INT}$ line signals that an input GPIO has changed and is usually attached to a microprocessor interrupt line. This saves the MCU from continuously polling the I2C devices to see if a button was pressed, for example. If any input line changes from high to low, or low to high, the open-drain transistor in the PCF8574 is activated and pulls the $\overline{INT}$ line low. This remains low until the device has its interrupt "serviced." A simple read or write to the peripheral is all that is necessary to service the interrupt.

The $\overline{INT}$ line does not identify which slave device has registered a change. The MCU must still poll its participating slave devices to see where the change occurred.

There is a small limitation that is important to keep in mind. If the GPIO level change occurs too quickly, no interrupt will be generated. It is also possible for a GPIO change event to occur during the ACK/NAK cycle when the interrupt is being cleared. An interrupt occurring then can also be lost. The NXP (NXP Semiconductors) datasheet indicates that it takes 4 µs from the sensing of a GPIO change to the activation of the $\overline{INT}$ line. The remaining time will consist of the MCU's interrupt response and software-handler processing.

PCF8574 Configuration

The NXP Semiconductors datasheet describes the I/O ports as *quasi-bidirectional*. What this means is that the GPIO ports (P0 through P7) can be used as outputs or be read as inputs directly, *without* any configuration through a device register.

To send an output value, you simply write to the PCF8574 device over the I2C bus. Input GPIOs, on the other hand, require a little trick—where you want GPIO *inputs*, you write a 1-bit to the GPIO port first. To see how this works, review Figure 11-7.
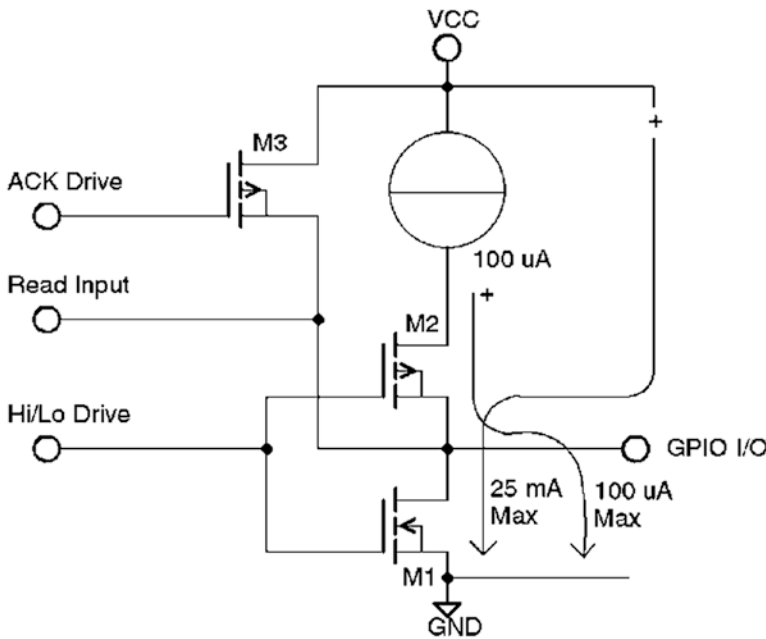
**Figure 11-7.** *PCF8574 simplified GPIO circuit*

Start at the top of the diagram where V_{cc} (+3.3 volts) is located. Within the chip is a 100 µA constant-current regulator in series with transistors M2 and M1. Consequently, when a high (1-bit) is written to the GPIO port, transistor M2 is turned on and M1 is turned off (driven by the "hi/lo drive" internal to the chip). Transistor M3 is off at this time.

If you short circuited the GPIO pin to ground, the constant-current regulator limits the current to 100 µA as it flows through M2 and out the GPIO pin to ground (rightmost arrow in Figure 11-7). While shorted like this, the internals of the PCF8574 are able to sense a low on the internal "Read Input" connected to the GPIO, which is read back as a 0-bit. By writing a 1-bit to the GPIO, you allow an external circuit to bring the voltage level low, or leave it pulled high. The current is always limited to a trickle of 100 µA, so no harm is done.

If, instead, the GPIO pin were written as a 0-bit, transistor M1 would always be turned on, shorting out the GPIO level. The "Read Input" would always be sensed as a 0-bit as a result. By the simple rule of writing a 1-bit to the GPIO, you can sense when it is pulled to ground as an input.

PCF8574 GPIO Drive

The quasi-bidirectional design of the GPIO has a consequence. You've already seen that the shorted GPIO output is current-limited to 100 μA. This means that the GPIO cannot act as a current *source* for an LED. A typical LED needs about 10 mA of current, which is 100 times what this GPIO is capable of supplying!
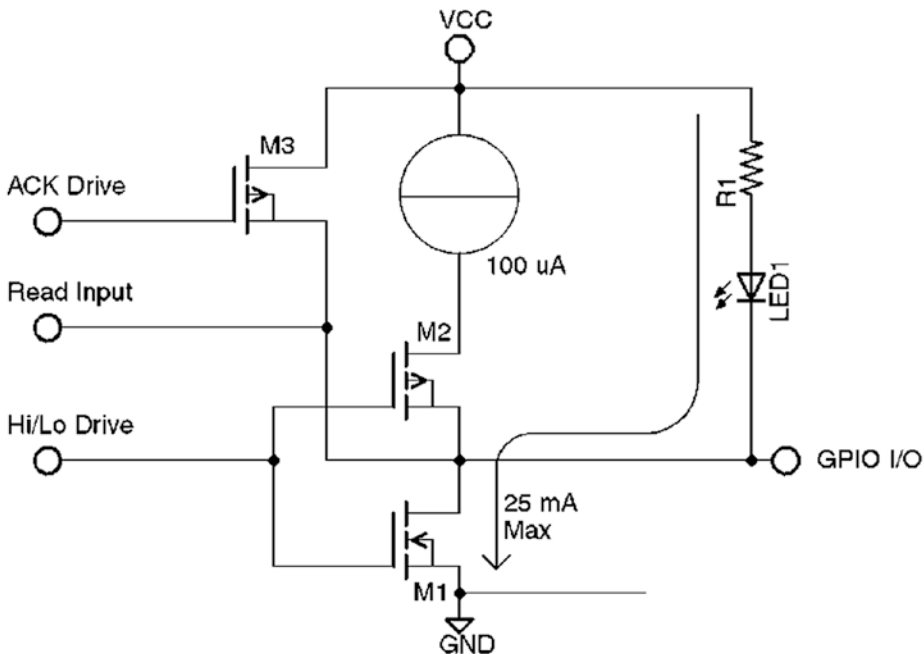
Figure 11-8. *Driving higher-current loads with the PCF8574*

However, transistor M1 is able to handle a maximum of 25 mA if you use the GPIO pin to *sink* power for the LED. Figure 11-8 illustrates how to drive an LED.

The LED and resistor R_1 are supplied from V_{cc}, which is not current limited. So, M1 acts as a switch to *sink* the current to ground, lighting the LED. The logic impact of this is that you need to write a 0-bit to turn the LED on.

Note   While the high drive of the PCF8574 is limited to 100 μA, this is sufficient for driving other CMOS (complementary metal oxide semiconductor) signal inputs.

Wave Shaping

When the GPIO output is written as a 1-bit, only 100 μA of current is available to pull it up to V_{cc}. This presents a bit of a problem when it is currently at low potential, resulting in a slow rise time.

The designers of the PCF8574 included a circuit with transistor M3 (Figure 11-8), which is normally off. However, when the device is written to, each GPIO that is receiving a 1-bit gets a boost from M3 during the I2C ACK/NAK cycle. This helps to provide a snappy low-to-high transition on the outputs. Once the ACK/NAK cycle is completed, M3 turns off again, leaving the 100 μA current limiter to maintain the high output.

Demo Circuit

Figure 11-9 illustrates the final circuit for the demo program. The noteworthy changes are that only one PCF8574 chip is used, using two LEDs and one push button.
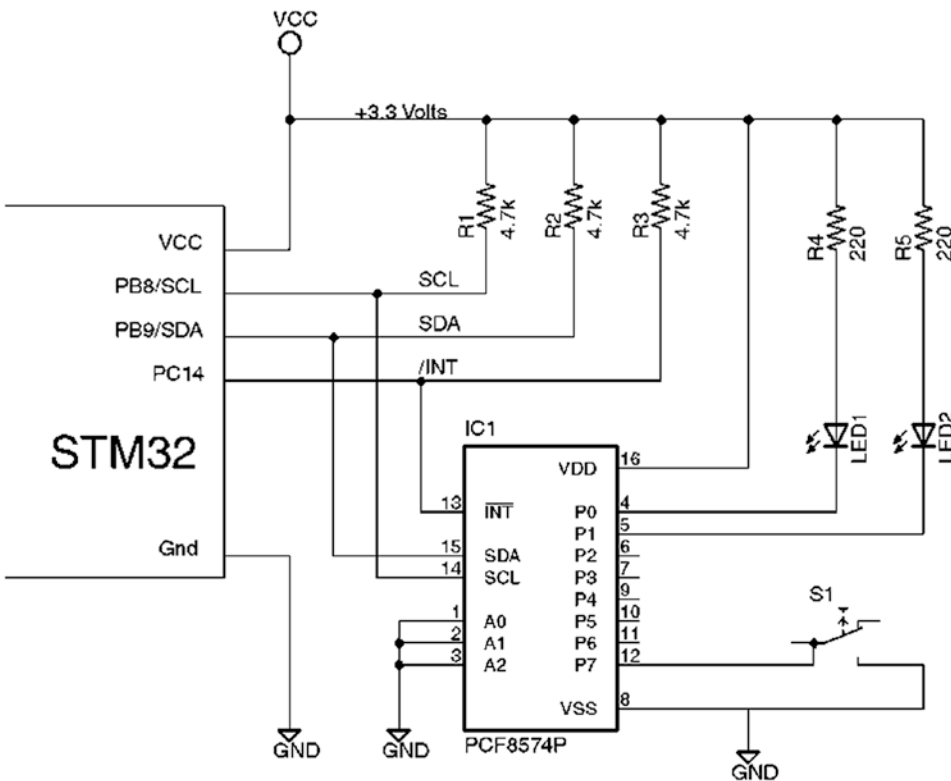
Figure 11-9. *The demo I2C schematic with LEDs and push button*

When wiring this circuit up, don't forget to include the pullup resistor R_3 so that the idle potential of GPIO PC14 is high. Note also that both LEDs are supplied from the V_{cc} rail so that ports P0 and P1 sink the current to light the LEDs. Port P7 will read the push button, which will be normally high (recall that the port is pulled high by the 100 µA constant-current source within the PCF8574). When the button is pressed, P7 will be pulled to ground, causing a 0-bit to be read.

EXTI Interrupt

The RTC2 project made use of the EXTI interrupt to achieve a separate alarm interrupt. A few more steps are required to achieve an interrupt on PC14 for the /INT interrupt. Let's look at the software involved. The project software for this chapter is found in this directory:

```
$ cd ~/stm32f103c8t6/rtos/i2c-pcf8574
```

Initially, we'll examine the I2C and EXTI setup in the main() routine of main.c (Listing 11-1).

Listing 11-1. The Initial Setup of the I2C Peripheral and EXTI Interrupts

```
0197: int
0198: main(void) {
0199:
0200:    rcc_clock_setup_in_hse_8mhz_out_72mhz();// For "blue pill"
0201:    rcc_periph_clock_enable(RCC_GPIOB);  // I2C
0202:    rcc_periph_clock_enable(RCC_GPIOC);  // LED
0203:    rcc_periph_clock_enable(RCC_AFIO);   // EXTI
0204:    rcc_periph_clock_enable(RCC_I2C1);   // I2C
0205:
0206:    gpio_set_mode(GPIOB,
0207:        GPIO_MODE_OUTPUT_50_MHZ,
0208:        GPIO_CNF_OUTPUT_ALTFN_OPENDRAIN,
0209:        GPIO6|GPIO7);                   // I2C
0210:    gpio_set(GPIOB,GPIO6|GPIO7);        // Idle high
0211:
```

```
0212:   gpio_set_mode(GPIOC,
0213:       GPIO_MODE_OUTPUT_2_MHZ,
0214:       GPIO_CNF_OUTPUT_PUSHPULL,
0215:       GPIO13);                          // LED on PC13
0216:   gpio_set(GPIOC,GPIO13);              // PC13 LED dark
0217:
0218:   // AFIO_MAPR_I2C1_REMAP=0, PB6+PB7
0219:   gpio_primary_remap(0,0);
0220:
0221:   gpio_set_mode(GPIOC,                  // PCF8574 /INT
0222:       GPIO_MODE_INPUT,                  // Input
0223:       GPIO_CNF_INPUT_FLOAT,
0224:       GPIO14);                          // on PC14
0225:
0226:   exti_select_source(EXTI14,GPIOC);
0227:   exti_set_trigger(EXTI14,EXTI_TRIGGER_FALLING);
0228:   exti_enable_request(EXTI14);
0229:   nvic_enable_irq(NVIC_EXTI15_10_IRQ);    // PC14 <- /INT
```

Lines 201 through 204 enable clocks that are needed by GPIOB (for I2C), GPIOC (for LED- and PC14-sensing $\overline{INT}$), EXTI, and the I2C peripheral itself. Line 206 configures GPIO PB6 and PB7 for open-drain operation for the I2C peripheral. Line 210 may not be strictly necessary, but until the I2C peripheral is configured, the I2C bus lines should be allowed to be pulled high.

Line 219 configures I2C1 to use PB6 and PB7. The gpio_primary_remap() libopencm3 function can be used to make other choices. PB6 and PB7 are extra useful because these have 5-volt-tolerant inputs.

Line 221 sets up GPIO PC14 to be an input in floating mode. This line will be pulled high by R_3 in Figure 11-9.

Lines 226 through 229 configure the EXTI interrupt. The exti_select_source() function chooses GPIO PC14 to be added to the list of potential interrupt sources. Line 227 then configures that the interrupt should occur when the signal falls from high to low. Finally, line 228 enables the EXTI peripheral to request interrupts. The call to nvic_enable_irq() enables the interrupt vector NVIC_EXTI15_10_IRQ. When this interrupt occurs, the entry point exti15_10_isr() will be called.

To save you much head scratching if you're working from the ST Microelectronics datasheet (RM0008), Table 11-2 is provided. The datasheet isn't clear, in my opinion, about how the interrupts are supported for EXTI. The table shows that lines zero through four have their own private interrupt vector. But for GPIO ports numbering 5 to 9, or 10 to 15, the interrupt vectors are more widely shared.

Table 11-2. *The List of EXTI STM32F103 Interrupts and ISR Routine Names*

Interrupt	ISR Routine	Description
NVIC_EXTI0_IRQ	exti0_isr()	Line 0: PA0/PB0/PC0
NVIC_EXTI1_IRQ	exti1_isr()	Line 1: PA1/PB1/PC1
NVIC_EXTI2_IRQ	exti2_isr()	Line 2: PA2/PB2/PC2
NVIC_EXTI3_IRQ	exti3_isr()	Line 3: PA3/PB3/PC3
NVIC_EXTI4_IRQ	exti4_isr()	Line 4: PA4/PB4/PC4
NVIC_EXTI9_5_IRQ	exti9_5_isr()	Lines 5 to 9: PA5-9/PB5-9/PC5-9
NVIC_EXTI15_10_IRQ	exti15_10_isr()	Lines 10 to 15: PA10-15/PB10-15/PC10-15
NVIC_PVD_IRQ	pvd_isr()	Line 16: Power
NVIC_RTC_ALARM_IRQ	rtc_alarm_isr()	Line 17: RTC Alarm
NVIC_USB_WAKEUP_IRQ	usb_wakeup_isr()	Line 18: USB Wakeup

I2C Software

The source code to drive the I2C peripheral has been placed in the source module `i2c.c`. The first function of interest is `i2c_configure()`, illustrated in Listing 11-2.

Listing 11-2. I2C Configuration

```
0061: void
0062: i2c_configure(I2C_Control *dev,uint32_t i2c,uint32_t ticks) {
0063:
0064:   dev->device = i2c;
0065:   dev->timeout = ticks;
0066:
0067:   i2c_peripheral_disable(dev->device);
```

```
0068:    i2c_reset(dev->device);
0069:    I2C_CR1(dev->device) &= ~I2C_CR1_STOP;     // Clear stop
0070:    i2c_set_standard_mode(dev->device);        // 100 kHz mode
0071:    i2c_set_clock_frequency(dev->device,I2C_CR2_FREQ_36MHZ); // APB Freq
0072:    i2c_set_trise(dev->device,36);             // 1000 ns
0073:    i2c_set_dutycycle(dev->device,I2C_CCR_DUTY_DIV2);
0074:    i2c_set_ccr(dev->device,180);              // 100 kHz <= 180 * 1
/36M
0075:    i2c_set_own_7bit_slave_address(dev->device,0x23);
0076:    i2c_peripheral_enable(dev->device);
0077: }
```

A structure named I2C_Control is passed in as the first argument to hold the configuration. The I2C peripheral address is passed in the argument i2c, which will be I2C1 for this demo. The last argument defines a timeout to be used, specified in ticks. These values are preserved in I2C_Control in lines 64 and 65 for later use.

Lines 67 to 69 clear and reset the I2C peripheral so that if it is stuck, it can be "unstuck." One of the disadvantages of I2C is that the protocol can sometimes hang if unpleasant things happen on the bus.

Lines 70 to 76 configure the I2C peripheral and enable it. Line 75 is only necessary if you want to operate the controller in slave mode.

Testing I2C Ready

Before any I2C operations can be initiated, you must test whether the device is busy. Otherwise, your request will likely be ignored or will impair the current operation. Listing 11-3 shows the routine used.

Listing 11-3. Testing for I2C Ready

```
0083: void
0084: i2c_wait_busy(I2C_Control *dev) {
0085:
0086:   while ( I2C_SR2(dev->device) & I2C_SR2_BUSY )
0087:       taskYIELD();              // I2C Busy
0088:
0089: }
```

This routine uses libopencm3 routines and macros to manage the peripheral. If the device is busy, however, the control is passed to other FreeRTOS tasks using the `taskYIELD()` statement.

Start I2C

To initiate an I2C bus transaction, the peripheral must perform a "start" operation. This can involve bus arbitration if there are multiple masters being used. Listing 11-4 shows the routine used by the demo.

Listing 11-4. I2C Start Function

```
0095: void
0096: i2c_start_addr(I2C_Control *dev,uint8_t addr,enum I2C_RW rw) {
0097:   TickType_t t0 = systicks();
0098:
0099:   i2c_wait_busy(dev);                    // Block until not busy
0100:   I2C_SR1(dev->device) &= ~I2C_SR1_AF; // Clear Acknowledge failure
0101:   i2c_clear_stop(dev->device);         // Do not generate a Stop
0102:   if ( rw == Read )
0103:       i2c_enable_ack(dev->device);
0104:   i2c_send_start(dev->device);          // Generate a Start/Restart
0105:
0106:   // Loop until ready:
0107:   while ( !((I2C_SR1(dev->device) & I2C_SR1_SB)
0108:     && (I2C_SR2(dev->device) & (I2C_SR2_MSL|I2C_SR2_BUSY))) ) {
0109:       if ( diff_ticks(t0,systicks()) > dev->timeout )
0110:           longjmp(i2c_exception,I2C_Addr_Timeout);
0111:       taskYIELD();
0112:   }
0113:
0114:   // Send Address & R/W flag:
0115:   i2c_send_7bit_address(dev->device,addr,
0116:       rw == Read ? I2C_READ : I2C_WRITE);
0117:
```

```
0118:    // Wait until completion, NAK, or timeout
0119:    t0 = systicks();
0120:    while ( !(I2C_SR1(dev->device) & I2C_SR1_ADDR) ) {
0121:        if ( I2C_SR1(dev->device) & I2C_SR1_AF ) {
0122:            i2c_send_stop(dev->device);
0123:            (void)I2C_SR1(dev->device);
0124:            (void)I2C_SR2(dev->device);      // Clear flags
0125:            // NAK Received (no ADDR flag will be set here)
0126:            longjmp(i2c_exception,I2C_Addr_NAK);
0127:        }
0128:        if ( diff_ticks(t0,systicks()) > dev->timeout )
0129:            longjmp(i2c_exception,I2C_Addr_Timeout);
0130:        taskYIELD();
0131:    }
0132:
0133:    (void)I2C_SR2(dev->device);          // Clear flags
0134: }
```

The first step is to determine the current tick time in line 97. This allows us to time the operation and time out if necessary. Line 99 waits for the peripheral to become ready. Once ready, line 100 clears an acknowledge failure, if there was one. Line 101 indicates that no stop should be generated.

If the operation is going to be a read, the i2c_enable_ack() is called to allow receipt of the ACK from the slave. The peripheral is then told to generate a start bit in line 104.

Lines 107 and 108 test if the start bit has been generated. If it is not yet generated, lines 109 and 110 test and perform a longjmp() if the operation has timed out. We'll speak more about the longjmp() acting as an exception later. If not timed out, the FreeRTOS statement taskYIELD() is performed to share the CPU while we wait.

Once the start bit has been generated, execution continues at line 115 to send the slave address and the read/write indicator.

In line 119 we note the time again for another potential timeout. Line 120 waits for the I2C address to be sent, while line 121 tests if the slave device ACKed the request. If no device responds to the address requested, a NAK will be received by default (thanks to the pull-up resistor). If the operation times out, a longjmp() is performed at line 129.

If the operation succeeds, a flag is cleared by calling I2C_SR2() to read the status register.

I2C Write

Once the start bit has been generated and the address sent, if we indicated that a write follows, we must do that next. Listing 11-5 shows the write function used.

Listing 11-5. I2C Write Function

```
0140: void
0141: i2c_write(I2C_Control *dev,uint8_t byte) {
0142:   TickType_t t0 = systicks();
0143:
0144:   i2c_send_data(dev->device,byte);
0145:   while ( !(I2C_SR1(dev->device) & (I2C_SR1_BTF)) ) {
0146:       if ( diff_ticks(t0,systicks()) > dev->timeout )
0147:           longjmp(i2c_exception,I2C_Write_Timeout);
0148:       taskYIELD();
0149:   }
0150: }
```

Line 142 notes the time for a possible timeout. Line 144 ships the data byte to the I2C peripheral to be sent serially on the bus. Line 145 tests for the completion of this operation and times out with a longjmp() if necessary (line 147). Aside from sharing the CPU with taskYIELD(), the function returns when successful.

I2C Read

If the intention was to read, the read routine is used to read a data byte. Listing 11-6 illustrates the code used.

Listing 11-6. The I2C Read Function

```
0157: uint8_t
0158: i2c_read(I2C_Control *dev,bool lastf) {
0159:   TickType_t t0 = systicks();
0160:
0161:   if ( lastf )
0162:       i2c_disable_ack(dev->device);    // Reading last/only byte
0163:
```

```
0164:    while ( !(I2C_SR1(dev->device) & I2C_SR1_RxNE) ) {
0165:        if ( diff_ticks(t0,systicks()) > dev->timeout )
0166:            longjmp(i2c_exception,I2C_Read_Timeout);
0167:        taskYIELD();
0168:    }
0169:
0170:    return i2c_get_data(dev->device);
0171: }
```

One of the unusual aspects of the i2c_read() function presented is that it has a Boolean lastf flag. This is set true by the caller if it is the *last* or *only* byte to be read. This gives the slave device a head's up that it can relax (some slaves must prefetch data in order to stay in step with the master controller). This is the purpose of the call on line 162.

Otherwise, it is a matter of status testing in line 164 and timing out in line 166 if the operation takes too long. Otherwise, the CPU is shared with taskYIELD(), and the byte is returned in line 170.

I2C Restart

The i2c_write_restart() routine partially shown in Listing 11-7 provides the ability to change from a write request into another request (read or write) without stopping. You can continue with the same slave device (by repeating the same slave address) or switch to another. This is significant when there are multiple I2C masters because this permits another message without renegotiating the access to the bus.

Listing 11-7. The "Secret Sauce" to Performing an I2C Restart Transaction

```
void
0179: i2c_write_restart(I2C_Control *dev,uint8_t byte,uint8_t addr) {
0180:    TickType_t t0 = systicks();
0181:
0182:    taskENTER_CRITICAL();
0183:    i2c_send_data(dev->device,byte);
0184:    // Must set start before byte has written out
0185:    i2c_send_start(dev->device);
0186:    taskEXIT_CRITICAL();
```

Some of this kind of information is difficult to tease from the STM32 reference manual (RM0008). However, careful attention to the fine print and footnotes can sometimes yield gold nuggets. The manual says:

> *In master mode, setting the START bit causes the interface to generate a ReStart condition at the end of the current byte transfer.*

By making lines 182 to 186 a critical section, you guarantee that you request another "start" prior to the current I2C's write being completed.

Demo Program

Listing 11-8 illustrates the main loop of the demo program. Most of it is straightforward, but there are a few things that are noteworthy. After the I2C device is configured in line 131, the inner loop begins at line 134. As long as there is no keyboard input, this loop continues writing and reading from the PCF8574 chip.

Listing 11-8. Main Loop of the Demo Program

```
0116: static void
0117: task1(void *args __attribute__((unused))) {
0118:   uint8_t addr = PCF8574_ADDR(0);    // I2C Address
0119:   volatile unsigned line = 0u;       // Print line #
0120:   volatile uint16_t value = 0u;      // PCF8574P value
0121:   uint8_t byte = 0xFF;               // Read I2C byte
0122:   volatile bool read_flag;           // True if Interrupted
0123:   I2C_Fails fc;                      // I2C fail code
0124:
0125:   for (;;) {
0126:       wait_start();
0127:       usb_puts("\nI2C Demo Begins "
0128:           "(Press any key to stop)\n\n");
0129:
0130:       // Configure I2C1
0131:       i2c_configure(&i2c,I2C1,1000);
0132:
```

```
0133:         // Until a key is pressed:
0134:         while ( usb_peek() <= 0 ) {
0135:             if ( (fc = setjmp(i2c_exception)) != I2C_Ok ) {
0136:                 // I2C Exception occurred:
0137:                 usb_printf("I2C Fail code %d\n\n",
0138:                                 fc,i2c_error(fc));
                     break;
0139:             }
0140:
0141:             read_flag = wait_event(); // Interrupt or timeout
0142:
0143:             // Left four bits for input, are set to 1-bits
0144:             // Right four bits for output:
0145:
0146:             value = (value & 0x0F) | 0xF0;
0147:             usb_printf("Writing $%02X "
0148: #if 0           "I2C @ $%02X\n",value,addr);
0149:             /*********************************************
0150:              * This example performs a write transaction,
0151:              * followed by a separate read transaction:
0152:              *********************************************/
0153:             i2c_start_addr(&i2c,addr,Write);
0154:             i2c_write(&i2c,value&0x0FF);
0155:             i2c_stop(&i2c);
0156:
0157:             i2c_start_addr(&i2c,addr,Read);
0158:             byte = i2c_read(&i2c,true);
0159:             i2c_stop(&i2c);
0160: #else
0161:             /*********************************************
0162:              * This example performs a write followed
0163:              * immediately by a read in one I2C transaction,
0164:              * using a "Repeated Start"
0165:              *********************************************/
```

```
0166:            i2c_start_addr(&i2c,addr,Write);
0167:            i2c_write_restart(&i2c,value&0x0FF,addr);
0168:            byte = i2c_read(&i2c,true);
0169:            i2c_stop(&i2c);
0170: #endif
0171:            if ( read_flag ) {
0172:                // Received an ISR interrupt:
0173:                if ( byte & 0b10000000 )
0174:                    usb_printf("%04u: BUTTON RELEASED: "
0175:                        "$%02X; wrote $%02X, "
                            "ISR %d\n",
0176:                        ++line,byte,
                            value,isr_count);
0177:                else  usb_printf("%04u: BUTTON PRESSED:   "
0178:                        "$%02X; wrote $%02X, "
                            "ISR %d\n",
0179:                        ++line,byte,
                            value,isr_count);
0180:            } else  {
0181:                // No interrupt(s):
0182:                usb_printf("%04u:             "
                        "Read:  $%02X, "
0183:                        "wrote $%02X, ISR %d\n",
0184:                        ++line,byte,value,isr_count);
0185:            }
0186:            value = (value + 1) & 0x0F;
0187:        }
0188:
0189:        usb_printf("\nPress any key to restart.\n");
0190:    }
0191: }
```

Of particular note is the setjmp() at line 135. Since C lacks the exception mechanism that C++ possesses, the longjmp() was used instead. Our doing a setjmp() at the top of the loop allows us to make several later I2C calls, each with its own points of failure,

including timeouts. If *any* failure occurs, the longjmp() will take the control back to line 135 and return a *non-zero* failure code. From there the problem can be reported and exited out of the inner loop.

The setjmp/longjmp mechanism does exact a small price, however. Notice that variables line, value, and read_flag are marked *volatile* (lines 119 to 122). This was necessary to silence the compiler because it warns about those values' being changed as a result of the longjmp(), should it occur. The setjmp saves a bunch of registers, while the longjmp restores them to bring control back. Any variables still cached in a register would be clobbered by a longjmp.

There are #if, #else, and #endif statements in lines 148, 160, and 170, respectively. By changing line 148 from the value zero to a non-zero value, all transactions will be individual; i.e., the byte will be written out to the PCF8574P in one transaction, followed by a completely separate I2C transaction to read from it.

Leaving line 148 at the value zero allows you to test the I2C restart operation. Lines 166 through 169 perform a write followed by a read in the same transaction.

Demo Session

Perform a build from scratch as follows:

```
$ make clobber
$ make
arm-none-eabi-gcc ... -o main.elf
arm-none-eabi-size main.elf
   text    data     bss     dec     hex    filename
  13024      28   18200   31252    7a14    main.elf
```

Ready the device for flashing and perform the following:

```
$ make flash
arm-none-eabi-objcopy -Obinary main.elf main.bin
/usr/local/bin/st-flash  write main.bin 0x8000000
...
2017-12-09T21:32:12 INFO src/common.c: Flash written and verified!
                                       jolly good!
```

Now, plug the USB cable in and start minicom, as follows:

```
$ minicom usb
Welcome to minicom 2.7

OPTIONS:
Compiled on Sep 17 2016, 05:53:15.
Port /dev/cu.usbmodemWGDEM1, 21:33:40

Press Meta-Z for help on special keys

Task1 begun.

Press any key to begin.
```

Once again, the "usb" argument to minicom is the name of the file that you saved your minicom settings to. I used the file named usb in this example.

Once you see "Task1 begun," press any key. I pressed Return. Once you do that, the I2C device should get configured, and you should start seeing messages of the following form:

```
I2C Demo Begins (Press any key to stop)

Writing $F0 I2C @ $20
0001:          Read:  $F0, wrote $F0, ISR 0
Writing $F1 I2C @ $20
0002:          Read:  $F1, wrote $F1, ISR 0
```

If you press a key again, the control will stop and then fall out to the outer loop. Pressing a key again will restart the demo in the inner loop.

The values written out to the PCF8574P will increment in the lower four bits. If you attached LEDs to P0 and P1 as in the schematic, you should see them count down in binary. When you press the button, you should see some messages indicating button press and release events.

```
Writing $F5 I2C @ $20
0006:          Read:  $F5, wrote $F5, ISR 0
Writing $F6 I2C @ $20
0007:          Read:  $F6, wrote $F6, ISR 0
```

```
Writing $F7 I2C @ $20
0008: BUTTON PRESSED:  $77; wrote $F7, ISR 4
Writing $F8 I2C @ $20
0009: BUTTON PRESSED:  $78; wrote $F8, ISR 4
Writing $F9 I2C @ $20
0010: BUTTON PRESSED:  $79; wrote $F9, ISR 5
Writing $FA I2C @ $20
0011: BUTTON PRESSED:  $7A; wrote $FA, ISR 6
Writing $FB I2C @ $20
0012: BUTTON PRESSED:  $7B; wrote $FB, ISR 7
Writing $FC I2C @ $20
0013: BUTTON PRESSED:  $7C; wrote $FC, ISR 8
Writing $FD I2C @ $20
0014:           Read:  $7D, wrote $FD, ISR 8
Writing $FE I2C @ $20
0015: BUTTON RELEASED: $FE; wrote $FE, ISR 11
```

The value shown after ISR shows you how many times the ISR routine was called when the PCF8574P indicated an interrupt. My button was pretty scratchy, and without any debouncing you see several button-press events. Notice that while the button was held down, the upper bit changed from a 1-bit to a 0-bit (for example, $FX changed to a $7X).

Summary

This chapter leaves you well prepared for I2C work. The PCF8574 is a very economical solution for adding more GPIO ports provided you don't have high speed requirements. At the same time, it provides you with experience in the world of I2C. The PCF8574 has demonstrated that it can generate interrupts so that you don't have to continually poll for input-line changes. This eases the burden of I2C traffic on the bus.

EXERCISES

1. What is the byte value sent when reading from slave address $21 (hexadecimal)?

2. When the master requests a response from a non-existing slave device on the bus, how does the NAK get received?

3. What is the advantage of the /INT line from the PCF8574?

4. What does *quasi-bidirectional* mean in the context of the PCF8574?

5. What is the difference between sourcing and sinking current?

OLED

The OLED (organic light-emitting diode) provides the hobbyist with an exciting form of low-cost display. Because they are based upon LEDs, they require no backlighting like an LCD device does, nor polarizing filters. This equates to lower cost.

The OLED device used in this chapter is monochrome, though it may display two colors in addition to black. That sounds contradictory, but the monochrome nature just means that it only displays one color for a given pixel. OLEDs with dual colors will have a band of pixels in one color, with the remainder in another. The background is always black (LED not lit).

The devices available today are small, usually 128 x 32 or 128 x 64 pixels in size. The physical dimensions also tend to be small. Yet because of their low cost and vivid color, they make great display widgets. This chapter will demonstrate the display of an analog meter on an OLED.

OLED Display

The unit I purchased from eBay was advertised as "White/Blue/Yellow Blue 0.96" SPI Serial 128 x 64 OLED LCD LED Display Module S" for a few dollars. But be wary of the "I2C" versus "SPI" in the listing. Many vendors don't get this right.

The important thing is that the OLED should use the SSD1306 controller for the demo software. The display itself is WiseChip part number UG-2864HSWEG01, although the auction might not state that. Some eBay offers may be selling display part number UG-2864AMBAG01, which is considerably different and can't be used with this chapter's software. If you don't mind paying a little more, Adafruit sells them as "Monochrome 0.96" 128 x 64 OLED graphic display." Buying from them is easier than trying to obtain the correct part from eBay.

Figure 12-1 illustrates the OLED display I am using. The Adafruit OLED is similar, but the backside of the PCB differs. This chapter's software requires a unit 128 pixels wide by 64 pixels high.
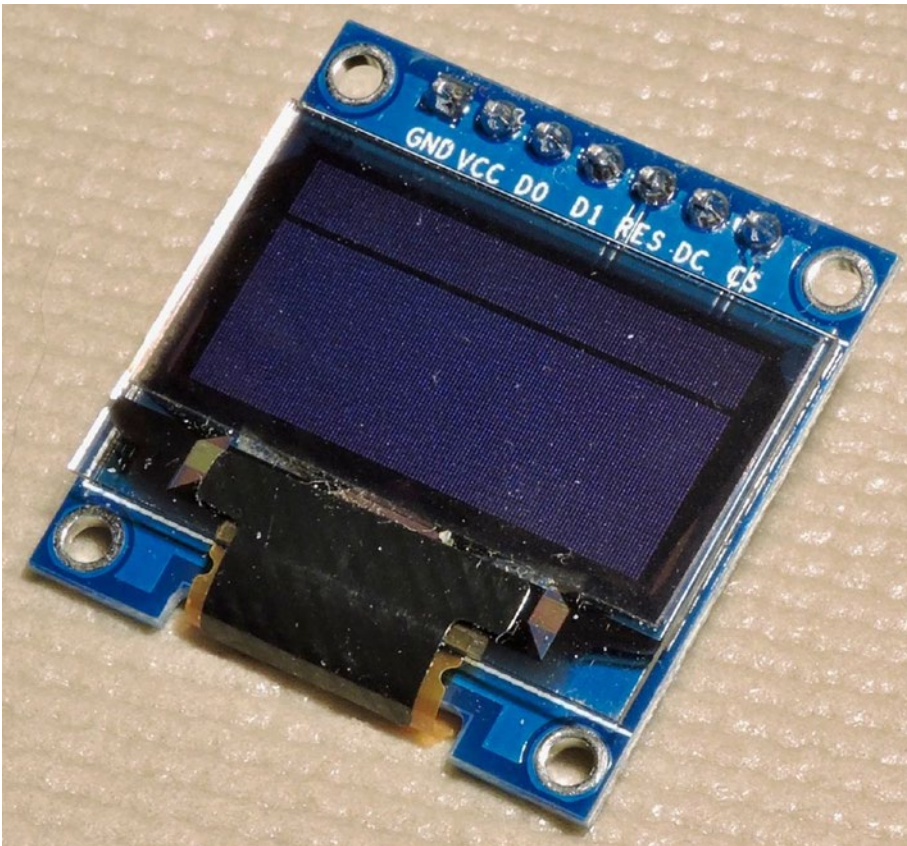
Figure 12-1. *The OLED using a SSD1306 controller*

Configuration

For this demo, you want a unit configured for *four-wire SPI*. The bottom side configures the device according to the resistors installed (Figure 12-2). Note that R_3 and R_4 are installed in the figure, confirming that this unit is configured for four-wire SPI. Those using the Adafruit unit should have jumper pads SJ1 and SJ2 unconnected.
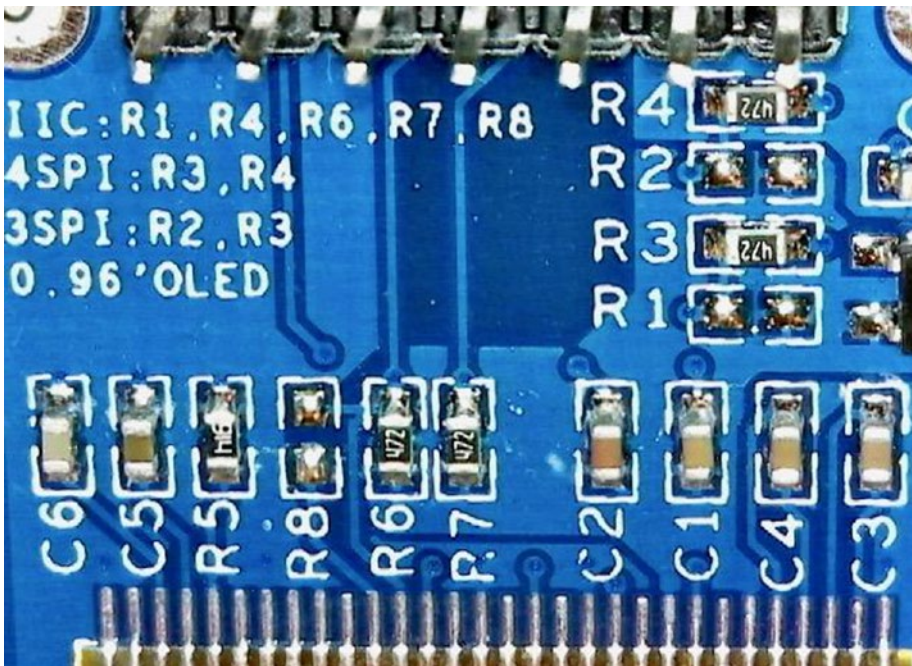
Figure 12-2. *The backside of the OLED, illustrating the configuration resistors R_1 through R_8*

Table 12-1 summarizes the different configurations possible. The four-wire SPI reference refers to the normal three SPI signals plus an additional line indicating a command or data signal. This extra line goes low to indicate when a command byte is being sent, and high for display data.

Table 12-1. *OLED Configurations*

R1	R2	R3	R4	Configuration
In	Out	Out	In	I2C (not used for this demo)
Out	Out	In	In	Four-wire SPI
Out	In	In	Out	Three-wire SPI (not used for this demo)

Display Connections

The display unit comes with seven connections, listed in Table 12-2. The $\overline{Reset}$ connection is *optional* and should be wired *high* (inactive) if unused. The demo program will use a GPIO to activate reset at startup.

The precise current draw will depend upon several factors. Adafruit suggests that typical current may be about 20 mA. In my own testing, I measured a current of 13.2 mA with all pixels on. But different OLED configuration options may increase current consumption. This level is low enough that it is safe to supply the OLED from the +3.3-volt regulator.

Table 12-2. *OLED Connections*

OLED Pin	Function	Description
Gnd	Ground	Common return path
VCC	3.3 to 5.0 volts	Supply voltage (up to 20 mA)
D0 (or SCK)	SCK	SPI system clock
D1 (or SDA)	SDIN	SPI MOSI (system data in for OLED)
RES	$\overline{Reset}$	Reset signal (active low)
DC	Data / $\overline{Command}$	Data (high), Command (low)
CS	$\overline{ChipSelect}$	Chip select (active low)

Display Features

Before examining the demo program, it is helpful to look at the OLED display features that it will be driving. Figure 12-3 illustrates the author's OLED with all pixels turned on (using controller command 0xA5).
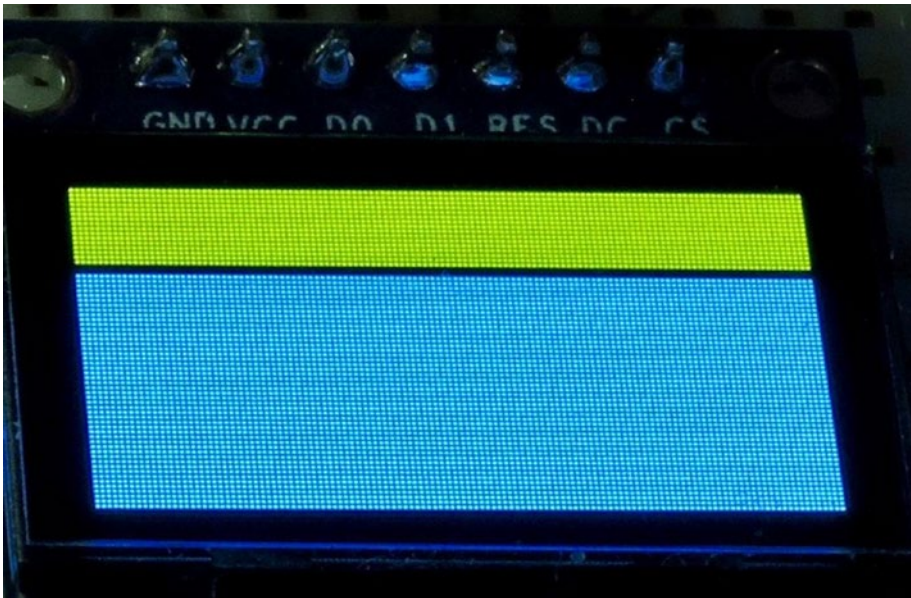
Figure 12-3. *Author's yellow/blue OLED with all pixels on*

While this is a yellow/blue OLED, the display is monochrome. You only get yellow in the top sixteen rows. After a gap of one row, there are forty-eight rows of blue pixels below. Some single-color units might lack this gap. Choose carefully for your application.

With all pixels turned on, my OLED measured 13.3 mA of current.

Demo Schematic

The demo circuit uses the same SPI hookup we used in the Winbond project (Chapter 8) but uses a few extra control lines for the OLED device. This demo still uses SPI1 for the SPI controller but is using an alternate GPIO configuration, to be described later. PA15 is acting as $\overline{NSS}$ that will drive the chip select of the OLED. PB10 will signal to the OLED whether commands or data are being sent. Finally, PB11 can be activated at startup to initialize the OLED when the demo program begins. Figure 12-4 illustrates the demo circuit.
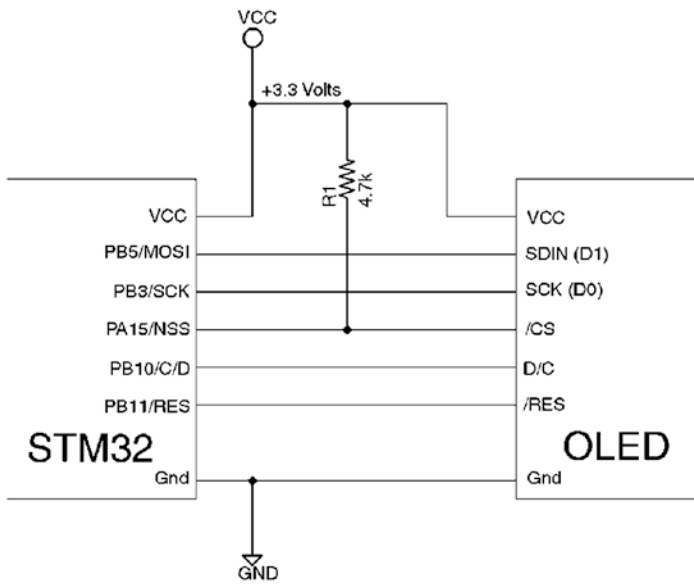
Figure 12-4. *Demo OLED circuit using SPI*

The $\overline{RES}$ pin of the OLED, which is wired to the SSD1306 controller, must remain low for a minimum of 3 μs for it to be effective. The reset sets the controller into several default modes, which saves some initialization.

AFIO

The STM32 platform supports the concept of remapping I/O functions. It is referred to in their documentation as "Alternate Function I/O." This chapter's demo takes advantage of this feature to have SPI1 appear on GPIOs PA15, PB3, PB4, and PB5. Table 12-3 lists the AFIO options for SPI1.

Table 12-3. *Alternate Function I/O for SPI1*

Alternate Function	SPI1_REMAP=0	SPI1_REMAP=1
SPI1_NSS	PA4	PA15
SPI1_SCK	PA5	PB3
SPI1_MISO	PA6	PB4
SPI1_MOSI	PA7	PB5

The AFIO feature allows additional flexibility in planning your STM32 resources. If you needed 5-volt-tolerant inputs, you would want to use SPI1_REMAP=1. Sometimes AFIO is used to avoid conflict with pins used by another peripheral.

To take advantage of AFIO, you need to get the following ducks in a row:

1. Enable the AFIO clock.

2. Configure the alternate function.

3. Configure GPIO *outputs* for ALTFN. *Inputs* do not require special treatment other than to be configured as an input.

All three of these steps are *essential*. Forgetting to enable the AFIO clock, for example, will result in nothing happening or the peripheral hanging. Using libopencm3, the AFIO clock is enabled with the following:

```
rcc_periph_clock_enable(RCC_AFIO);
```

The demo program uses the following libopencm3 call to choose SPI1's alternate function using libopencm3's gpio_primary_remap() function:

```
// Put SPI1 on PB5/PB4/PB3/PA15
gpio_primary_remap(
    AFIO_MAPR_SWJ_CFG_JTAG_OFF_SW_OFF, // Optional
    AFIO_MAPR_SPI1_REMAP);             // SPI1_REMAP=1
```

The first argument disables JTAG functionality and is secondary to our goal of remapping. The second argument indicates that you want SPI1 to be remapped (SPI_REMAP=1 in Table 12-3). The natural mapping (SPI1_REMAP=0) is used by default after a system reset.

For GPIO *outputs*, you *must* choose one of the following macros when configuring it. Otherwise, the peripheral would not be able to reach the output pins.

- GPIO_CNF_OUTPUT_ALTFN_PUSHPULL

- GPIO_CNF_OUTPUT_ALTFN_OPENDRAIN

For example:

```
 gpio_set_mode(
     GPIOB,
     GPIO_MODE_OUTPUT_50_MHZ,
     GPIO_CNF_OUTPUT_ALTFN_PUSHPULL,  // Note!
     GPIO5|GPIO3);
```

Notice the ALTFN in the argument three macro name. An easy mistake to make is to use the non-ALTFN macro instead and then wonder why your peripheral is not communicating with the pin.

Graphics

One of the hurdles when working with graphics devices is performing operations like drawing lines, circles, and rectangles. It is true that lines and rectangles are simple enough if they use perfectly horizontal and vertical lines. But lines tilted on an angle and filled circles present a challenge. Then, there is the need for fonts.

These software problems are large enough that the average developer doesn't want to expend time on re-developing solutions for them. After all, these are problems that have been solved before. Why do we have to keep solving them again?

The good news is that the problem has been solved before and that the software is available in *open source* form. The demo project in this chapter will employ the graphics software written by Achim Döbler, available on github here:

```
https://github.com/achimdoebler/UGUI
```

The one characteristic of this graphics software that I give the author kudos for is that it is designed to be adapted to any graphics platform. Aside from some simple configuration in the ugui_config.h file, the only other requirement is a user-supplied function:

```
void
local_draw_point(UG_S16 x,UG_S16 y,UG_COLOR c) {
        ...
}
```

Given x and y coordinates and a color, this function is called upon to draw a point in your own graphics environment. To make this work, the uGUI environment is simply initialized with a function pointer:

```
static UG_GUI gui;
    ...
    UG_Init(&gui,local_draw_point,128,64);
```

The arguments 128 and 64 in this example define the maximum width and height of the drawing canvas. Once this has been done, uGUI functions can be called upon to fill a circle; for example:

```
    UG_FillCircle(x,y,c);
```

The demo project is located in the following directory:

```
$ cd ~/stm32f103c8t6/rtos/oled
```

Our OLED device, however, is monochrome, so some special color handling is needed. To translate color into monochrome, the following routine is provided in meter.c:

```
0059: static int
0060: ug_to_pen(UG_COLOR c) {
0061:
0062:   switch ( c ) {
0063:   case C_BLACK:
0064:       return 0;
0065:   case C_RED:
0066:       return 2;
0067:   default:
0068:       return 1;
0069:   }
0070: }
```

This function merely converts any color except for red and white to a 1 (white), with black represented as a 0. The color red is used by the demo software to represent exclusive-or.

The exclusive-or operation has the special property that if the pixel is currently 0 (black) it will be painted as white. If the current pixel is white, then it is converted to black. Regardless of the current state of the graphics canvas, something is always visibly drawn in exclusive-or mode.

> **Tip**   Depending upon your compiler and options used, you may want to spend
> time reducing the amount of code compiled. Use #if to eliminate unused
> functions in the ugui.c module.

The Pixmap

To facilitate graphic drawing on the OLED, a pixel map (pixmap) buffer is used. This
allows extensive drawing operations to occur at full CPU speed. At the appropriate time,
the pixmap is then copied to the OLED device for display.

The pixmap is defined in the file meter.c as follows:

```
static uint8_t pixmap[128*64/8];
```

This defines 128 times 64 pixels, with eight pixels to a byte, thus using 1024 bytes of
SRAM.

To facilitate drawing into the pixmap, the to_pixel() function is used, illustrated in
Listing 12-1. It computes a byte address within the pixmap based upon the given x and y
coordinates and then returns a bit number through the pointer argument bitno.

Listing 12-1. The to_pixel() Function

```
0020: static uint8_t dummy;
0021:
0022: static uint8_t *
0023: to_pixel(short x,short y,unsigned *bitno) {
0024:    *bitno = 7 - y % 8;       // Inverted
0025:
0026:    if ( x < 0 || x >= 128
0027:      || y < 0 || y >= 64 )
0028:          return &dummy;
0029:
0030:    unsigned inv_y = 63 - y;
0031:    unsigned pageno = inv_y / 8;
0032:    unsigned colno = x % 128;
```

```
0033:
0034:    return &pixmap[pageno * 128 + colno];
0035: }
```

A couple of points require explaining. Lines 24 and 30 are used to invert the display. This was done to arrange that the yellow band of 16 rows would appear as the top of the display. To use non-inverted coordinates, you would change line 24 from:

```
0024:    *bitno = 7 - y % 8;        // Inverted
```

to

```
0024:    *bitno = y % 8;            // Non-inverted
```

Likewise, y would be used instead of the computed inv_y value. Centralizing this mapping in one place makes it possible to introduce translations to the display. For example, you could rework this function to transform x and y to display on the device in portrait mode rather than landscape.

In theory, there should be no call to this routine with the x and y coordinates out of range. But should that happen, the routine returns a pointer to the value dummy so that the call can be ignored without fatal consequences.

Pixmap Writing

After the byte and bit numbers have been determined by the to_pixel() function, the actual point-drawing function becomes simpler, shown in Listing 12-2. The draw_point() function is called by the earlier local_draw_point() function. The draw_point() routine expects the 2, 1, or 0 pen value rather than a color.

Listing 12-2. The Internal draw_point() Function

```
0037: static void
0038: draw_point(short x,short y,short pen) {
0039:
0040:    if ( x < 0 || x >= 128 || y < 0 || y >= 64 )
0041:        return;
0042:
0043:    unsigned bitno;
```

```
0044:    uint8_t *byte = to_pixel(x,y,&bitno);
0045:    uint8_t mask = 1 << bitno;
0046:
0047:    switch ( pen ) {
0048:    case 0:
0049:        *byte &= ~mask;
0050:        break;
0051:    case 1:
0052:        *byte |= mask;
0053:        break;
0054:    default:
0055:        *byte ^= mask;
0056:    }
0057: }
```

Lines 40 and 41 exit the function without doing anything when the x and/or y coordinates are out of range. Otherwise, lines 43 and 44 determine the byte address and bit number for the pixel being altered. Line 45 computes a bit mask from the bitno value and saves it to mask.

What happens next depends upon the pen value. If the pen was 0 (white), that bit is masked out so that the pixel bit is cleared to zero. If the pixel is 1, the mask value is or-ed with the byte to produce a 1-bit in the pixel. Finally, in line 55, the default pen value (2 normally) will produce an exclusive-or of the pixel instead.

The Meter Software

The graphics software specific to the meter display is found in the file meter.c. Those interested in the design of this program can examine the source code for the details. For brevity, I'll just highlight the important functions within it.

meter_init()

```
void meter_init(struct Meter *m,float range);
```

If this were C++, you could think of the meter_init() function as the constructor. The struct Meter m is initialized by the call, while the float argument range configures the meter's upper range. In the demo main.c program, range is provided as 3.5 for 3.5 volts.

meter_set_value()

```
void meter_set_value(struct Meter *m,float v);
```

This function changes the value stored in meter object m to the value v. This will move the graphics pointer in the pixmap.

meter_redraw()

```
void meter_redraw(struct Meter *m);
```

This function is used internally at initialization time to draw the entire meter into the pixmap. It can be called again if the software suspects or knows that the image was corrupted somehow. In the demo, this is only called once at initialization.

meter_update()

This is the function used to transfer the pixmap in SRAM to the OLED using SPI1:

```
void meter_update(void);
```

The SPI transfer code is illustrated in Listing 12-3.

Listing 12-3. The meter_update() SPI Transfer Function

```
0195: void
0196: meter_update(void) {
0197:    uint8_t *pp = pixmap;
0198:
0199:    oled_command2(0x20,0x02);// Page mode
0200:    oled_command(0x40);
0201:    oled_command2(0xD3,0x00);
0202:    for ( uint8_t px=0; px<8; ++px ) {
0203:        oled_command(0xB0|px);
0204:        oled_command(0x00); // Lo col
0205:        oled_command(0x10); // Hi col
0206:        for ( unsigned bx=0; bx<128; ++bx )
0207:            oled_data(*pp++);
0208:    }
0209: }
```

Line 197 obtains the address of the first byte of the pixmap. Line 199 makes certain that the SSD1306 controller is in "page mode." In this mode, the OLED memory is broken up into eight pages of 128 bytes of pixels.

Line 200 initializes the SSD1306 to start at display line zero, while line 201 initializes the SSD1306 to set the display offset to zero.

The loop in lines 202 to 208 then takes care of transferring data one page at a time to the OLED. Line 203 chooses the OLED page to update. Lines 204 and 205 initialize the column index to zero. Lines 206 and 207 actually pass the data to the OLED controller and update the display pixel data pointer pp.

The functions oled_command(), oled_command2(), and oled_data() are found in the demo module main.c.

Main Module

Since the OLED module requires some special processing with the Data/Command signal line, let's examine the functions used by the meter program.

oled_command()

This function is used to send command bytes to the OLED controller and is illustrated in Listing 12-4.

Listing 12-4. The oled_command() Function

```
0034: void
0035: oled_command(uint8_t byte) {
0036:   gpio_clear(GPIOB,GPIO10);
0037:   spi_enable(SPI1);
0038:   spi_xfer(SPI1,byte);
0039:   spi_disable(SPI1);
0040: }
```

Line 36 clears GPIO PB10 so that the Data/Command line goes *low*, indicating to the OLED controller that SPI data is to be interpreted as command bytes. Lines 37 to 39 transfer this command byte over SPI1.

oled_command2() is identical, except that it sends two command bytes instead of one.

oled_data()

The oled_data() function is very similar to oled_command(). It simply sets the GPIO line PB10 high (line 53 of Listing 12-5) so that the OLED controller will accept SPI data as pixel data.

Listing 12-5. The oled_data() Function

```
0051: void
0052: oled_data(uint8_t byte) {
0053:   gpio_set(GPIOB,GPIO10);
0054:   spi_enable(SPI1);
0055:   spi_xfer(SPI1,byte);
0056:   spi_disable(SPI1);
0057: }
```

oled_reset()

The main module calls upon function oled_reset() to initialize the OLED controller, as shown in Listing 12-6.

Listing 12-6. The oled_reset() Function

```
0059: static void
0060: oled_reset(void) {
0061:   gpio_clear(GPIOB,GPIO11);
0062:   vTaskDelay(1);
0063:   gpio_set(GPIOB,GPIO11);
0064: }
```

Line 61 sets PB11 to low. Then, FreeRTOS routine vTaskDelay() is called for one tick (about 1 ms), which should be more than enough time (a minimum of 3 µs is required). Then, after the delay in line 62, the PB11 pin is brought high again.

oled_init()

The function oled_init is illustrated in Listing 12-7. Lines 73 and 77 are non-essential, simply activating the built-in LED on PC13. The OLED is reset in line 74 and is followed by several commands sent to it from the array cmds (line 68) from the loop in lines 75 and 76.

Listing 12-7. The oled_init() Function

```
0066: static void
0067: oled_init(void) {
0068:   static uint8_t cmds[] = {
0069:       0xAE, 0x00, 0x10, 0x40, 0x81, 0xCF, 0xA1, 0xA6,
0070:       0xA8, 0x3F, 0xD3, 0x00, 0xD5, 0x80, 0xD9, 0xF1,
0071:       0xDA, 0x12, 0xDB, 0x40, 0x8D, 0x14, 0xAF, 0xFF };
0072:
0073:   gpio_clear(GPIOC,GPIO13);
0074:   oled_reset();
0075:   for ( unsigned ux=0; cmds[ux] != 0xFF; ++ux )
0076:       oled_command(cmds[ux]);
0077:   gpio_set(GPIOC,GPIO13);
0078: }
```

Demonstration

In the project directory, perform the following:

```
$ make clobber
$ make
$ make flash
```

Once your STM32 is flashed and wired up according to the schematic in Figure 12-4, you should be able to unplug the programmer and then plug in the USB cable for the STM32 device. After a brief pause, you should see the display in Figure 12-5, if everything is working. Depending upon your device, you may see different colors.

Tip   When developing a new project, if the linker tells you that `.bss` will not fit in region `ram`, review the value of `configTOTAL_HEAP_SIZE` in file `FreeRTOSConfig.h`. You may need to reduce the heap size to make room for your program's own storage.
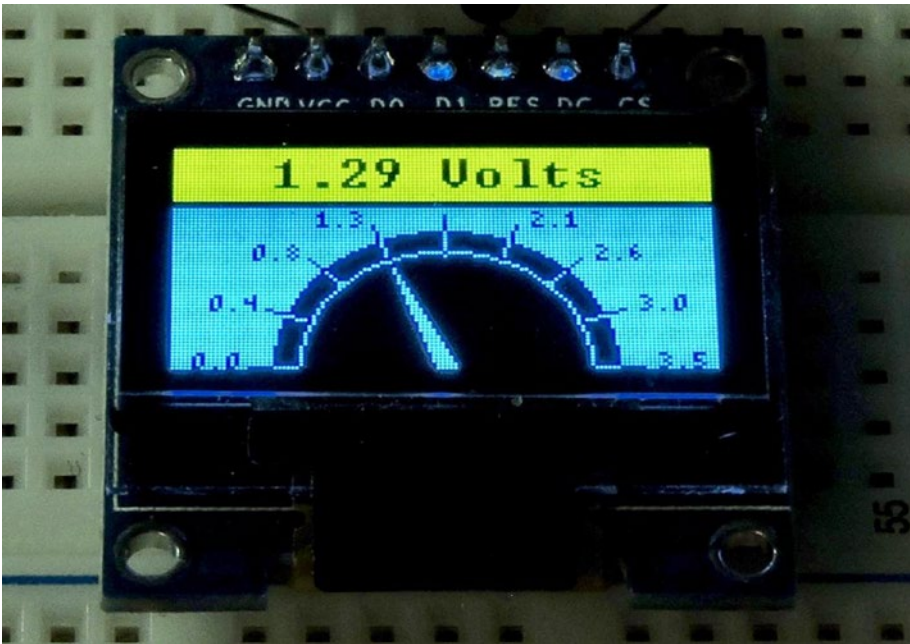
Figure 12-5. *The demonstration program produces a voltmeter graphic on the OLED*

If the display did not initialize correctly, it is best to immediately unplug the USB cable and recheck your wiring. If successful, start up minicom with your USB startup settings (mine is named "usb"):

```
$ minicom usb
```

After minicom connects to your USB device and starts, you should see a session display like the following:

```
Welcome to minicom 2.7

OPTIONS:
Compiled on Sep 17 2016, 05:53:15.
Port /dev/cu.usbmodemWGDEM1, 22:46:21

Press Meta-Z for help on special keys
```

Press the `Return` key to prompt a menu display from the demo program:

```
Test Menu:
  0 .. set to 0.0 volts
  1 .. set to 1.0 volts
  2 .. set to 2.0 volts
  3 .. set to 3.0 volts
  4 .. set to 3.5 volts
  + .. increase by 0.1 volts
  - .. decrease by 0.1 volts

: _
```

Pressing "1" should immediately cause the meter (OLED) to display 1.0 volts. Likewise, pressing "3" points the meter at 3 volts. Pressing the "+" or "-" key will allow you to increase/decrease respectively the voltage displayed by tenths of a volt.

Summary

In this chapter, SPI was applied to the real-world problem of driving an OLED display. In doing so, the advantage of using open-sourced software for graphics operations was demonstrated. Graphics permitted the drawing of an analog meter on the OLED as well as the use of a font to display the voltage digitally.

Moreover, the signals for Data/ $\overline{Command}$ and $\overline{RESET}$ were demonstrated to drive the OLED display, in addition to the usual SPI signals.

The concept of AFIO for the STM32 family was also applied in this chapter to demonstrate how SPI1 could have its I/O pins moved to different pins. AFIO permits greater flexibility in applying the resources of the STM32 chip.

EXERCISES

1. For AFIO output pins, what GPIO configuration macros must be used?

2. What clock must be enabled for AFIO changes?

3. What GPIO configuration macros should be used for input pins?

4. What is the purpose of the OLED D/C input?

OLED Using DMA

In the previous chapter, software was developed to drive the OLED using master-mode SPI transactions. The STM32 platform does, however, support a DMA (direct memory access) controller, which can be exploited to perform the I/O operations and leave more cycles available for the CPU. This chapter will explore how to set up and use that DMA controller to drive the OLED device.

Challenges

This project will challenge us a little bit because our OLED device requires some special handling. The main challenges are as follows:

- The OLED SSD1306 controller allows us to only update one of eight pages at a time, requiring multiple DMA transfers.

- In between pages of data sent, additional SSD1306 controller *commands* must be sent to select the next page to be updated.

- Switching between OLED *commands* and *data* requires us to change a GPIO signal level in between transfers, which cannot be integrated with the DMA operation itself.

In some applications, it is possible to configure the DMA controller and simply launch it. The DMA controller then optionally notifies us of completion with an interrupt. In this project, the DMA will be launched a number of times to refresh eight pages of OLED memory data. This project will allow you to learn how to conquer this challenge.

Circuit

The circuit used for this chapter's project is identical to the one used in Chapter 12 (see Figure 12-4). The changes for this project are entirely contained within the software used.

DMA Operation

The DMA controller is a simple machine that operates in three phases:

1. Initial configuration

2. Execution

3. Notification of completion, or repeat execution

The DMA's purpose is to read data from its source and write it to its destination. How exactly this is performed depends upon its configuration.

DMA Execution

How does the DMA controller manage this automated data transfer? In this chapter, the presented project will configure the DMA controller to read data from one of two memory locations:

- array of OLED command bytes, or

- array of pixel data bytes.

As far as the DMA controller is concerned, our data source will be *memory*. When configuring the controller, the software will configure the byte address and length.

The DMA destination will be the SPI data port for transmitting (a *peripheral*). The configured destination therefore will be the memory-mapped port address for SPI1's data register (&SPI1_DR in C-language terms).

There is still the matter of *when* a given data byte is transferred. A DMA transfer cycle consists of the following series of events:

1. A request signal is sent to the DMA controller.

2. The DMA controller performs the data transfer (in this case, memory to peripheral). The configured priority will determine which transfer occurs first.

3. The DMA controller sends an *acknowledge* signal to the requestor.

4. The request signal is *released*.

5. The DMA acknowledge signal is *released*.

6. The DMA controller decrements the length count.

7. When configured to do so, the source or destination address is incremented. The address is incremented by the size (in bytes) of the transfer.

This process repeats until the transfer length reaches zero. At that point, the DMA controller reaches a *completed* status, which will include a completion interrupt when configured for it.

DMA Request Signals

Internal to the STM32 MCU are request signals connected to *DMA channels*. The STM32F103C8T6 MCU is a *medium-density* controller and thus has only one DMA controller (DMA1). DMA1 supports seven DMA channels, while larger MCUs sport a second controller that supports an additional five channels.

Table 13-1 summarizes the DMA channels supported by the STM32F103C8T6. Our project will make use of DMA1, channel 3 because it supports the requestor SPI1_TX.

Table 13-1. *Supported DMA1 Channels*

Requestor	Channel	Description
ADC1	1	DMA1, Channel 1 (highest priority)
TIM2_CH3		
TIM4_CH1		
USART3_TX	2	DMA1, Channel 2
TIM1_CH1		
TIM2_UP		
TIM3_CH3		
SPI1_RX		
USART3_RX	3	DMA1, Channel 3
TIM1_CH2		
TIM3_CH4		
TIM3_UP		
SPI1_TX		

(*continued*)

Table 13-1. (*continued*)

Requestor	Channel	Description
USART1_TX	4	DMA1, Channel 4
TIM1_CH4		
TIM1_TRIG		
TIM1_COM		
TIM1_CH2		
SPI2/I2S2_RX		
I2C2_TX		
USART1_RX	5	DMA1, Channel 5
TIM1_UP		
SPI2/I2C2_TX		
TIM2_CH1		
TIM4_CH3		
I2C2_RX		

(*continued*)

Table 13-1. (*continued*)

Requestor	Channel	Description
USART2_RX	6	DMA1, Channel 6
TIM1_CH3		
TIM3_CH1		
TIM3_TRIG		
I2C1_TX		
USART2_TX	7	DMA1, Channel 7 (lowest priority)
TIM2_CH2		
TIM2_CH4		
TIM4_UP		
I2C1_RX		

Because of the groupings in Table 13-1, it is evident that other signals can act on channel 3 in addition to SPI1_TX. These are as follows:

- USART3_RX
- TIM1_CH2
- TIM3_CH4
- TIM3_UP

Only *one* of these requestors can be active at a time. An application needing to use SPI1_TX and USART3_RX must arrange it so that the DMA controller is only configured for one of these at a given instant.

246

Each channel has a configured priority of four levels. However, if competing channels have the same priority, the *lowest* numbered channel has the priority.

You can think of Table 13-1 as the wiring between peripherals and the DMA controller request lines. For example, SPI1 can request for transmission on channel 3, while its receiving requests are wired to channel 2. SPI2 is hardwired to request on channels 4 and 5.

A *memory-to-memory* transfer can also be performed by the DMA controller, with source and destination on any available channel.

SPI1_TX Request

Our project will make use of the SPI1_TX request, available on DMA channel 3. This request line is active when the following are true:

- The SPI1 status register SPI_SR flag TXE flag is set to 1 (transmit buffer empty).

- The SPI1 control register SPI_CR2 flag TXDMAEN is set to 1 (DMA enabled).

- The SPI1 peripheral itself is enabled (register SPI_CR1 bit SPE set to 1).

Assuming the remaining aspects of SPI1 configuration are correct, establishing the preceding three conditions activates the DMA request line. In order for the DMA controller to respond to this, the following conditions must also be met:

- One-time configuration of DMA has been established.

- The DMA channel is enabled.

Once those conditions are established in both the SPI1 peripheral and the DMA controller, then the DMA operation will proceed without software intervention.

The Demonstration

Seeing the involved software will help to bring these concepts into focus. The source code for this chapter is included in the following directory:

```
$ cd ~/stm32f103c8t6/rtos/oled_dma
```

Change to that directory and rebuild from scratch:

```
$ make clobber
$ make
$ make flash
```

Note   It is usually necessary to change the Boot0 jumper to flash the device when a prior flash has configured AFIO. Set Boot0=1, leave Boot1=0, and then flash. Return Boot0=0.

Listing 13-1 summarizes a few changes made to the main() program from the previous chapter's source code. Line 403 affects the SPI I/O transfer rate. With the divisor set to 64, the SPI SCLK rate is increased to 1.125 Mhz. If you experience trouble getting your circuit to work, increase the divisor to 256. Breadboard arrangements can be very noisy and limit performance.

Listing 13-1. Main Program Changes

```
0401:   spi_init_master(
0402:           SPI1,
0403:           SPI_CR1_BAUDRATE_FPCLK_DIV_64, // 1.125 MHz
0404:           SPI_CR1_CPOL_CLK_TO_0_WHEN_IDLE,
0405:           SPI_CR1_CPHA_CLK_TRANSITION_1,
0406:           SPI_CR1_DFF_8BIT,
0407:           SPI_CR1_MSBFIRST
0408:   );
...
0412:   // DMA
0413:   rcc_periph_clock_enable(RCC_DMA1);
0414:   nvic_set_priority(NVIC_DMA1_CHANNEL3_IRQ,0);
0415:   nvic_enable_irq(NVIC_DMA1_CHANNEL3_IRQ);
...
0422:   xTaskCreate(spidma_task,"spi_dma",100,NULL,1,&h_spidma);
```

For the DMA1 operation, line 413 enables a system clock. Lines 414 and 415 configure the NVIC (nested vectored interrupt controller) to allow generation of the DMA1 channel 3 operation-complete interrupt.

Finally, line 422 creates another FreeRTOS task `spidma_task()` to orchestrate the DMA transfers needed.

Listing 13-2 illustrates a small change made to the `meter.c` module. It simply calls into the `main.c` module to issue an OLED update request in line 199.

Listing 13-2. Modification to meter.c

```
0197: void
0198: meter_update(void) {
0199:   spi_dma_xmit_pixmap();
0200: }
```

Listing 13-4 (later) illustrates the `spi_dma_xmit_pixmap()` function, which gets the OLED DMA I/O started.

Initializing DMA

There is a fair amount of software required to get the DMA controller set up for use. The good news is that much of it only needs to be done once. Listing 13-3 illustrates the one-time DMA configuration used by the demonstration program.

Listing 13-3. One-time DMA Initialization

```
0189: static void
0190: dma_init(void) {
0191:
0192:   dma_channel_reset(DMA1,DMA_CHANNEL3);
0193:   dma_set_peripheral_address(DMA1,DMA_CHANNEL3,
              (uint32_t)&SPI1_DR);
0194:   dma_set_read_from_memory(DMA1,DMA_CHANNEL3);
0195:   dma_enable_memory_increment_mode(DMA1,DMA_CHANNEL3);
0196:   dma_set_peripheral_size(DMA1,DMA_CHANNEL3,DMA_CCR_PSIZE_8BIT);
0197:   dma_set_memory_size(DMA1,DMA_CHANNEL3,DMA_CCR_MSIZE_8BIT);
0198:   dma_set_priority(DMA1,DMA_CHANNEL3,DMA_CCR_PL_HIGH);
0199:   dma_enable_transfer_complete_interrupt(DMA1,DMA_CHANNEL3);
0200: }
```

Line 192 resets the DMA1 controller. This clears the controller of any fault conditions and establishes a number of convenient defaults. Line 193 specifies the peripheral address to be SPI1_DR (SPI1 data register). Line 194 indicates that the controller will be reading from memory for channel 3 (thus the peripheral will be the written destination). Line 195 configures the DMA controller to increment the memory address after each byte is transferred. Line 196 indicates that the unit size is the byte for the peripheral, while the next line does the same for the memory side. Line 198 gives DMA channel 3 a high priority. Line 199 enables notifications of the DMA transfer completion by interrupt.

At this stage, the DMA1 controller is poised for action, needing just a few more details before it can pounce.

Launching DMA

The first step in launching the OLED refresh by DMA uses the main.c routine spi_dma_xmit_pixmap(), illustrated in abbreviated form in Listing 13-4. When DMA is started for the first time, this routine calls function start_dma(). We'll discuss the full logic of that routine later on.

Listing 13-4. Starting the DMA Transfer

```
0156: void
0157: spi_dma_xmit_pixmap(void) {
...
0169:   if ( prime )
0170:       start_dma();      // Start from idle
0171: }
```

The code for start_dma() is provided in Listing 13-5. The first thing it does is reset the OLED pageno value back to zero (line 146) and save the start of the OLED pixmap buffer in pointer variable pixmapp (line 147).

The first SPI I/O requires command bytes, so the GPIO PB10 is set to low to indicate to the OLED controller that the following data are command bytes (line 148). Finally, the spidma_task() is "goosed" in line 149.

Listing 13-5. Initiating the spidma Task to Start a New DMA Transfer

```
0040: static TaskHandle_t h_spidma = NULL;
...
0044: static volatile uint8_t *pixmapp = NULL;
0045: static volatile uint8_t pageno = 0;
...
0142: static void
0143: start_dma(void) {
0144:   extern uint8_t pixmap[128*64/8];
0145:
0146:   pageno = 0;
0147:   pixmapp = &pixmap[0];
0148:   gpio_clear(GPIOB,GPIO10); // Cmd mode
0149:   xTaskNotifyGive(h_spidma);
0150: }
```

OLED SPI/DMA Management Task

The management of the OLED DMA I/O transfers is tricky because we must break the refresh into eight OLED page updates, each requiring its own set of command and data bytes. The task spidma_task() is shown in Listing 13-6.

Listing 13-6. The spidma_task() Managing OLED DMA Updates

```
0041: static volatile bool dma_busy = false;
0042: static volatile bool dma_idle = true;
0043: static volatile bool dma_more = false;
0044: static volatile uint8_t *pixmapp = NULL;
0045: static volatile uint8_t pageno = 0;
...
0088: static void
0089: spidma_task(void *arg __attribute((unused))) {
0090:   static uint8_t cmds[] = {
0091:       0x20, 0x02,   // 0: Page mode
0092:       0x40,         // 2: Display start line
0093:       0xD3, 0x00,   // 3: Display offset
```

```
0094:        0xB0,           // 5: Page #
0095:        0x00,           // 6: Lo col
0096:        0x10            // 7: Hi Col
0097:    };
0098:
0099:    for (;;) {
0100:        // Block until ISR notifies
0101:        ulTaskNotifyTake(pdTRUE,portMAX_DELAY);
0102:        if ( dma_busy ) {
0103:            spi_clean_disable(SPI1);
0104:            dma_busy = false;
0105:            if ( gpio_get(GPIOB,GPIO10) ) {
0106:                // Advance data
0107:                pixmapp += 128;
0108:                ++pageno;
0109:            }
0110:            // Toggle between Command/Data
0111:            gpio_toggle(GPIOB,GPIO10);
0112:        }
0113:
0114:        if ( pageno >= 8 ) {
0115:            // All OLED pages sent:
0116:            dma_idle = true;
0117:            if ( dma_more ) {
0118:                // Restart update
0119:                dma_more = false;
0120:                start_dma();
0121:            }
0122:        } else  {
0123:            // Another page to send:
0124:            cmds[5] = 0xB0 | pageno;
0125:            if ( !gpio_get(GPIOB,GPIO10) ) {
0126:                // Send commands:
0127:                if ( !pageno )
0128:                    spi_dma_transmit(&cmds[0],8);
0129:                else  spi_dma_transmit(&cmds[5],3);
```

```
0130:            } else {
0131:                // Send page data:
0132:                spi_dma_transmit(pixmapp,128);
0133:            }
0134:        }
0135:    }
0136: }
```

The task consists of executing a loop starting at line 99, looping forever. The first step in each loop is to call uITaskNotifyTake(), which causes it to block forever until notified. The DMA complete interrupt will notify the task in the ISR routine (to be examined shortly). Once notified, execution returns to line 102.

The dma_busy flag is examined in line 102 to see if a DMA operation is currently in progress. The first time through, however, control resumes at line 114. This checks for an OLED update complete, which it is not on the first time through. Control then passes to lines 124 and 125. Line 125 checks the status of GPIO PB10. If the state of PB10 is *low*, then we are sending out *command* bytes this time through. Line 124 has placed the correct pageno value into the command sequence (pageno=0 the first time through).

The first command sequence is longer (line 128) because additional commands are included to make sure the OLED SSD1306 controller is in the correct *update* mode. This is done in case a data error has caused the SSD1306 controller to execute an erroneous command at some point. Following those commands, the bytes at cmds[5] through cmds[7] are sent to establish the graphics page being updated. On pages 1 through 7, we simply send the page-setting commands alone for efficiency.

After the DMA has completed the command bytes send, the control passes to line 102 in the loop again, with dma_busy true. Line 103 performs a call to libopencm3 routine spi_clean_disable() to wait until it is safe to manipulate SPI1 after a DMA transfer. The DMA will have sent an SPI byte, but the data byte may not have left the SPI controller yet. When control reaches line 104, it is known to be safe to start a new I/O.

Line 105 checks the state of GPIO PB10. After the command bytes have been sent, this will still be *low*, causing lines 107 to 108 to be skipped this time around. Line 111 will be executed, however, changing the D/C line to *high* for a data transfer.

Control passes to line 125, but GPIO PB10 is high at this point, so line 132 executes. This launches a DMA SPI transfer of 128 bytes of *data*.

Around we go, and we end up at line 103 again. This time, the GPIO PB10 is high, so the pointer variable pixmapp is incremented by 128 (line 107) to point to the next graphics page. The pageno value is also incremented (line 108). Because of the toggle that happens in line 111, GPIO PB10 returns to *low*, indicating another few bytes of commands to be sent in lines 124 to 129.

This cycle repeats seven more times to cause pages 1 through 7 to be sent to the OLED controller. Eventually, pageno is incremented to 8, and this is noticed in line 114. Line 116 sets the flag dma_idle=true but will start yet another round of OLED updates if flag variable dma_more was found enabled. The reason for this check will be explained later.

The task-notify mechanism has been used to facilitate this transfer. To understand why, the ISR routine will now be revealed.

DMA ISR Routine

Listing 13-7 presents the DMA1 channel 3 ISR routine. Line 57 checks for the DMA1 channel 3 DMA_TCIF flag (transfer complete interrupt flag), and if so, clears it in the following line. As configured in this demo, this should be the only reason to enter this function.

Listing 13-7. The DMA Complete ISR

```
0053: void
0054: dma1_channel3_isr(void) {
0055:   BaseType_t woken __attribute__((unused)) = pdFALSE;
0056:
0057:   if ( dma_get_interrupt_flag(DMA1,DMA_CHANNEL3,DMA_TCIF) )
0058:       dma_clear_interrupt_flags(DMA1,DMA_CHANNEL3,DMA_TCIF);
0059:
0060:   spi_disable_tx_dma(SPI1);
0061:
0062:   // Notify spidma_task to start another:
0063:   vTaskNotifyGiveFromISR(h_spidma,&woken);
0064: }
```

The clearing of the DMA_TCIF flag is important in line 58. There are two other sources of interrupts possible in the same ISR. The complete list for a given DMA channel includes the following:

- DMA_TCIF — Transfer Complete Interrupt Flag

- DMA_TEIF — Transfer Error Interrupt Flag

- DMA_THIF — Transfer Half-done Interrupt Flag

The latter two are not used in the demo. For your reference, Table 13-2 lists all of the interrupts and ISR routine names available for DMA1.

Table 13-2. *Interrupt Vectors for DMA*

DMA1 **Channel**	**ISR Routine**
NVIC_DMA1_CHANNEL1_IRQ	dma1_channel1_isr
NVIC_DMA1_CHANNEL2_IRQ	dma1_channel2_isr
NVIC_DMA1_CHANNEL3_IRQ	dma1_channel3_isr
NVIC_DMA1_CHANNEL4_IRQ	dma1_channel4_isr
NVIC_DMA1_CHANNEL5_IRQ	dma1_channel5_isr
NVIC_DMA1_CHANNEL6_IRQ	dma1_channel6_isr
NVIC_DMA1_CHANNEL7_IRQ	dma1_channel7_isr

Line 60 of the ISR disables SPI1's requests for DMA, while line 63 notifies the spidma_task() function, which is blocked in a call to ulTaskNotifyTake(). Note that the ISR must use the "FromISR" version of the call vTaskNotifyGiveFromISR(). ISRs are limited in what they can do, so these special forms of the calls allow for that.

Restarting DMA Transfers

Now, let's present the spi_dma_xmit_pixmap()routine in full in Listing 13-8.

Listing 13-8. The Full Listing of the spi_dma_xmit_pixmap() Function

```
0156: void
0157: spi_dma_xmit_pixmap(void) {
0158:   bool prime = false;
0159:
0160:   taskENTER_CRITICAL();
0161:   if ( !dma_idle ) {
```

```
0162:        // Restart dma at DMA completion
0163:        dma_more = true;// Restart upon completion
0164:    } else  {
0165:        prime = true;   // Start from idle
0166:    }
0167:    taskEXIT_CRITICAL();
0168:
0169:    if ( prime )
0170:        start_dma();    // Start from idle
0171: }
```

If meter updates were to occur frequently enough, they might arrive faster than the OLED can be refreshed with DMA. With the SPI clock set for 1.125 MHz, the full OLED refresh requires about 7.54 ms. This demo doesn't have any provision for interrupting the DMA transfer after it begins, and it would be undesirable to leave the display partially written anyway. So, how do we handle this crunch?

When updates occur frequently, we don't want to interrupt the one in progress. However, once the current DMA transfer completes we want at least one more OLED update to occur in order to display the current state. The volatile flag variable dma_more serves this purpose. But we have a race condition to contend with.

Line 160 begins a critical section that cannot be interrupted. Interrupts include preemption to allow other tasks to be run. Disabling interrupts in line 160 allows a test of the current state of dma_idle. If the variable is found to be false, then it knows that a set of DMA transfers is in progress or coming to an end. In this case, dma_more is set to true to request one more OLED update when the current one completes.

However, if dma_idle is found to be true, the DMA machinery is known to be idle and must be started up again. The local flag variable prime is set to true in this case (line 165).

Executing the Demo

The demo executes the same as in Chapter 12. However, the interactive menu has a new option, "p"—the meter "pummel" command:

```
$ minicom usb
```

When minicom connects to your USB device, press Return to get it started:

```
Welcome to minicom 2.7

OPTIONS:
Compiled on Sep 17 2016, 05:53:15.
Port /dev/cu.usbmodemWGDEM1, 21:29:29

Press Meta-Z for help on special keys

Monitor Task Started.

Test Menu:
  0 .. set to 0.0 volts
  1 .. set to 1.0 volts
  2 .. set to 2.0 volts
  3 .. set to 3.0 volts
  4 .. set to 3.5 volts
  + .. increase by 0.1 volts
  - .. decrease by 0.1 volts
  p .. Meter pummel test

:
```

The menu items work as they did in the previous chapter, with menu option "p" added. This "pummel test" hits the meter with rapid updates. When activated by pressing "p," the meter will move from end to end in rapid updates. The code for the pummel test is illustrated in Listing 13-9.

Listing 13-9. The Pummel Test Routine

```
0230: static void
0231: pummel_test(struct Meter *m1) {
0232:   TickType_t t0 = xTaskGetTickCount();
0233:   double v = 0.0;
0234:   double incr = 0.05;
0235:
0236:   meter_set_value(m1,v);
0237:   meter_update();
0238:   while ( (xTaskGetTickCount() - t0) < 5000 ) {
```

```
0239:        vTaskDelay(6);
0240:        v += incr;
0241:        if ( v > 3.3 ) {
0242:            incr = -0.05;
0243:            v = 3.3;
0244:        } else if ( v < 0.0 ) {
0245:            v = 0.0;
0246:            incr = 0.05;
0247:        }
0248:        meter_set_value(m1,v);
0249:        meter_update();
0250:    }
0251: }
```

The test routine is designed to operate for five seconds (line 238). The delay in line 239 determines how quickly the meter is updated. Here it is set to delay for six ticks (about 6 ms) between updates. Given the update takes 7.54 ms, this will overlap with a DMA transfer at least some of the time.

You may find that the text part of the display does not get updated during the pummel test. It will catch up after the pummeling ends. This illustrates the nature of the problem.

Further Challenges

While the demonstration code works as intended, it has one remaining flaw. If the updates occur too frequently—say, for example, at one-millisecond intervals—the display shown on the OLED can lose the pointer. Why does this happen?

The background of the meter is only written to the pixmap once. Only the pointer and the digital reading are redrawn in the pixmap. Whenever the meter is moved, the original pointer is drawn in the background color to erase it, followed by writing the new pointer in the foreground color. What can happen is that the pixmap being copied by DMA to the OLED copies the erased pointer because of poor timing.

To correct for this, a few different approaches are possible. One approach would be to have the pixmap copied to another pixmap buffer using a DMA memory-to-memory transfer. Then, the OLED can be updated by this pixmap buffer, which is never modified by the ongoing software. This obviously involves extra time for the in-memory copy as well as another pixmap buffer in SRAM.

Another approach might be to limit the meter updates to a maximum frequency by software. After all, a hardware meter pointer is unable to update instantly. This just a taste of some of the problems that come up in embedded computing.

Summary

This chapter built upon the software developed in Chapter 12, adding the DMA controller to manage data transfers to the OLED device. The demo helped you develop familiarity with the DMA controller and its capabilities. Using a FreeRTOS task mechanism, the DMA transfer was managed with command and data transfers that occurred by manipulating GPIO line PB10. Finally, the DMA transfer-complete interrupt was used to knit the events together.

The use of DMA is not always this complicated. However, this demo prepares you for something more difficult than your average textbook example.

EXERCISES

1. In the demo program, what DMA controller aspects had to be changed before starting the next transfer?

2. Does each DMA channel have its own ISR routine?

3. In a memory-to-peripheral transfer, like the demo, where does the DMA request come from?

4. In the demo program where SPI was used, what were the three conditions necessary before a DMA transfer could begin?

Analog-to-Digital Conversion

Embedded computing often needs to convert an analog signal level into a digital form for analysis. One application is measuring temperature by the voltage developed across a semiconductor. It is no surprise then that the STM32 platform has both an analog-to-digital converter (ADC) and a built-in channel to the ADC for measuring temperature.

This chapter's demonstration project will illustrate how to use the libopencm3 routines to access the ADC peripheral, reading analog channels PA0 and PA1, in addition to reading the chip temperature and its internal reference voltage.

STM32F103C8T6 Resources

The STM32F103C8T6 sports two ADC controllers, specified by the following libopencm3 macro names:

- ADC1 — 12-bit Analog Digital Controller 1 with 18 input channels
- ADC2 — 12-bit Analog Digital Controller 2 with 16 input channels

These each support 16 analog input channels. ADC1 can also access internal levels for temperature and a reference voltage V_{ref}.

The ADC peripheral also includes a programmable prescaler that establishes the conversion rate. The input to the prescaler is the PCLK2 (same as APB2) clock. Since our demo initializes with the call

```
 rcc_clock_setup_in_hse_8mhz_out_72mhz();
```

this results in the APB2 frequency being established as

```
rcc_apb2_frequency = 72000000;
```
or 72 MHz. The ADC input clock must not exceed 14 MHz, so this limits us to the divisor 6, generating a clock of $72 \div 6 = 12$ MHz.

Demonstration

There is no schematic for this demonstration since all is provided by the onboard ADC peripheral. The only external connections of interest are the analog inputs PA0 and PA1. However, a schematic will be provided later for how to hook up a potentiometer to generate voltages that can be sensed. This chapter is mostly about how to arrange the software to operate the ADC peripheral.

Caution   GPIO inputs PA0 and PA1 are not 5-volt tolerant and should only receive voltages between zero and +3.3 volts.

The software for this chapter is found at the following directory:

```
$ cd ~/stm32f103c8t6/rtos/adc
```

Change to that subdirectory and rebuild the project from scratch:

```
$ make clobber
$ make
$ make flash
```

Tip   It should not be necessary to change the Boot-0 jumper to reflash the STM32 for this project, except perhaps the first time.

Analog Inputs PA0 and PA1

In the `main()` program, the ADC peripheral and its GPIOs are initialized. The first step configures the analog inputs, as follows:

```
0087:   rcc_periph_clock_enable(RCC_GPIOA);  // Enable GPIOA for ADC
0088:   gpio_set_mode(GPIOA,
0089:       GPIO_MODE_INPUT,
0090:       GPIO_CNF_INPUT_ANALOG,            // Analog mode
0091:       GPIO0|GPIO1);                     // PA0 & PA1
```

As usual, the clock for GPIO is enabled in line 87. Lines 88 to 91 configure GPIOs PA0 and PA1 for analog input. Notice that the value `GPIO_CNF_INPUT_ANALOG` is used to configure the GPIO input. This permits a *varying* voltage to reach the ADC instead of a digital high/low value.

ADC Peripheral Configuration

The main complexity of this demonstration is correctly configuring the ADC peripheral. The STM32 has a dizzying array of options in this area. Listing 14-1 illustrates the configuration used by this demo. All source code presented in this chapter is found in file `main.c`.

The ADC peripheral's clock needs to be turned on, which line 103 accomplishes. The ADC peripheral's power (not its clock) is disabled in line 104 for initialization.

Listing 14-1. ADC Configuration

```
0102:   // Initialize ADC:
0103:   rcc_peripheral_enable_clock(&RCC_APB2ENR,RCC_APB2ENR_ADC1EN);
0104:   adc_power_off(ADC1);
0105:   rcc_peripheral_reset(&RCC_APB2RSTR,RCC_APB2RSTR_ADC1RST);
0106:   rcc_peripheral_clear_reset(&RCC_APB2RSTR,RCC_APB2RSTR_ADC1RST);
0107:   rcc_set_adcpre(RCC_CFGR_ADCPRE_PCLK2_DIV6);// Set. 12MHz, Max. 14MHz
0108:   adc_set_dual_mode(ADC_CR1_DUALMOD_IND);       // Independent mode
0109:   adc_disable_scan_mode(ADC1);
0110:   adc_set_right_aligned(ADC1);
0111:   adc_set_single_conversion_mode(ADC1);
```

```
0112:    adc_set_sample_time(ADC1,ADC_CHANNEL_TEMP,ADC_SMPR_SMP_239DOT5CYC);
0113:    adc_enable_temperature_sensor();
0114:    adc_power_on(ADC1);
0115:    adc_reset_calibration(ADC1);
0116:    adc_calibrate_async(ADC1);
0117:    while (adc_is_calibrating(ADC1));
```

Lines 105 and 106 reset the ADC further. These are separate calls because of the different registers involved.

ADC Prescaler

Line 107 sets the ADC prescaler to operate at 12 MHz maximum. The ADC clock will function up to 14 MHz, but the divisor of 4 results in 18 MHz, which is clearly over the limit. If your application requires the highest possible ADC conversion rate, the only choice is to alter the CPU and other clocks first. Keep in mind that there are clock constraints affecting the USB controller, which may limit your options if USB is required.

ADC Modes

Lines 108 through 111 configure a series of different modes available. Line 108 allows ADC1 and ADC2 to be operated independently. Line 109 disables the scan-mode option, while line 110 configures the ADC to store the result right-justified in its register. Finally, line 111 configures the ADC to stop when a single conversion is completed:

```
0108:    adc_set_dual_mode(ADC_CR1_DUALMOD_IND);      // Independent mode
0109:    adc_disable_scan_mode(ADC1);
0110:    adc_set_right_aligned(ADC1);
0111:    adc_set_single_conversion_mode(ADC1);
```

Sample Time

Lines 112 and 113 establish the sample time to be used on the temperature and V_{ref} channels:

```
0112:    adc_set_sample_time(ADC1,ADC_CHANNEL_TEMP,ADC_SMPR_SMP_239DOT5CYC);
0113:    adc_set_sample_time(ADC1,ADC_CHANNEL_VREF,ADC_SMPR_SMP_239DOT5CYC);
0114:    adc_enable_temperature_sensor();
```

Each channel of the ADC can be sampled with a different number of clock cycles. The default is to have each conversion occur in 1.5 cycles (`ADC_SMPR_SMP_1DOT5CYC`). The total number of clock cycles is given by the following equation:

$$T_{conv} = SampleRate + 12.5$$

In the case of line 112, the conversion time for temperature requires the following:

$$T_{conv} = 239.5 + 12.5 \, cycles$$
$$= 252 \, cycles$$

Since the ADC clock rate is 12 MHz, we know that the total time for conversion is as follows:

$$T_{conv} = \frac{252}{12e6} + 21 \, \mu s$$

The default sample rate for a given channel is 1.5 cycles. Table 14-1 lists the sample rates that are available.

Table 14-1. *ADC Sample Rates*

libopencm3 Macro Name	Cycles	Total time (12 MHz ADC clock)
ADC_SMPR_SMP_1DOT5CYC	1.5 + 12.5 = 14	1.167 µs
ADC_SMPR_SMP_7DOT5CYC	7.5 + 12.5 = 20	1.667 µs
ADC_SMPR_SMP_13DOT5CYC	13.5 + 12.5 = 26	2.167 µs
ADC_SMPR_SMP_28DOT5CYC	28.5 + 12.5 = 41	3.417 µs
ADC_SMPR_SMP_41DOT5CYC	41.5 + 12.5 = 54	4.500 µs
ADC_SMPR_SMP_55DOT5CYC	55.5 + 12.5 = 68	5.667 µs
ADC_SMPR_SMP_71DOT5CYC	71.5 + 12.5 = 84	7.000 µs
ADC_SMPR_SMP_239DOT5CYC	239.5 + 12.5 = 252	21.00 µs

Readying the ADC

Before the ADC controller is used, three more steps are required (from Listing 14-1):

```
0114:    adc_power_on(ADC1);
0115:    adc_reset_calibration(ADC1);
0116:    adc_calibrate_async(ADC1);
0117:    while (adc_is_calibrating(ADC1));
```

Power is turned on by line 114, and calibration constants reset in line 115. Line 116 starts the calibration, while line 117 waits for this to complete. In the demonstration program this is all performed before the FreeRTOS scheduler is started.

Demonstration Run

Once the STM32 has been flashed with the demonstration code, plug in its USB cable and start minicom (again, "usb" is the file name that I used to save the USB comms parameters):

```
$ minicom usb

Welcome to minicom 2.7

OPTIONS:
Compiled on Sep 17 2016, 05:53:15.
Port /dev/cu.usbmodemWGDEM1, 12:38:28

Press Meta-Z for help on special keys

Temperature 24.72 C, Vref 1.19 Volts, ch0 1.92 V, ch1 0.00 V
Temperature 24.72 C, Vref 1.19 Volts, ch0 1.94 V, ch1 0.00 V
Temperature 24.72 C, Vref 1.19 Volts, ch0 1.97 V, ch1 0.00 V
Temperature 24.72 C, Vref 1.19 Volts, ch0 1.98 V, ch1 0.00 V
```

Every 1.5 seconds a new line will be displayed, showing the following:

- Internal STM32 temperature in °C (24.72 °C in the example)

- Internal V_{ref} value of the STM32 (1.19 volts in example)

- Channel 0 voltage (1.92 volts in first example line)

- Channel 1 voltage (0.00 volts in the example)

To test analog inputs A0 and A1, you can attach a jumper wire first to Gnd and then to your +3.3-volt supply. Do *not* apply voltages higher than that or *negative* voltages—*this could result in permanent damage*.

The session just shown had PA1 floating, while PA0 was grounded. If you now apply +3.3 volts to the PA0 input, the reported value should be close to +3.3 volts. I got 3.29 volts when I tried this. Repeat the grounding and +3.3-volt test on PA1, and the program should report identical results.

Reading ADC

With the ADC configured in the main() program, it is possible for the demo_task() function to read the analog voltages by channel, as follows:

```
0060:    int adc0, adc1;
...
0068:    adc0 = read_adc(0) * 330 / 4095;
0069:    adc1 = read_adc(1) * 330 / 4095;
```

To avoid floating point for speed and to reduce flash size, integer arithmetic is used here to compute voltages in variables adc0 and adc1. A 12-bit ADC result has 4096 possible steps, resulting in the returned result ranging from 0 to 4095. The result of the calculation when multiplied by 330 is volts times one hundred.

The software responsible for reading from the ADC is given in Listing 14-2.

Listing 14-2. Reading the ADC Results

```
0030: static uint16_t
0031: read_adc(uint8_t channel) {
0032:
0033:    adc_set_sample_time(ADC1,channel,ADC_SMPR_SMP_239DOT5CYC);
0034:    adc_set_regular_sequence(ADC1,1,&channel);
0035:    adc_start_conversion_direct(ADC1);
0036:    while ( !adc_eoc(ADC1) )
0037:        taskYIELD();
0038:    return adc_read_regular(ADC1);
0039: }
```

Line 33 sets the sample time to 21 μs, and the sampling sequence is established in line 34. It specifies that one channel is to be sampled (argument 2), with the channel given by the list, starting with argument address &channel (this is a one-element list).

Line 35 launches the successive approximation ADC in line 35 and waits for completion in lines 36 and 37. Once again, taskYIELD() is called to allow other tasks to efficiently share the CPU time. Finally, line 38 fetches the conversion result and returns it to the caller.

Computing Temperature

The demo_task() makes the following call to acquire internal temperature:

```
0066:    temp100 = degrees_C100();
```

Listing 14-3 lists the function degrees_C100().

Listing 14-3. The degrees_C100() Function

```
0044: static int
0045: degrees_C100(void) {
0046:    static const int v25 = 143;
0047:    int vtemp;
0048:
0049:    vtemp = (int)read_adc(ADC_CHANNEL_TEMP) * 3300 / 4095;
0050:
0051:    return (v25 - vtemp) / 45 + 2500;
          // temp = (1.43 - Vtemp) / 4.5 + 25.00
0052: }
```

The STM32F103C8T6 documentation is very sketchy about this calculation. Since the PDF reference document (RM0008) applies to a whole family of STM32 devices, it is difficult to sort out the calculation needed by the Blue Pill device.

The information needed is available from the PDF found at:

http://www.st.com/resource/en/datasheet/stm32f103tb.pdf

For the STM32F103x8 and STM32F103xB series chips, look at the PDF's Table 50. That table is made available as Table 14-2 in this chapter for your convenience.

As indicated in the source code, and derived from the PDF document, the temperature for the STM32F103C8T6 device is computed as follows:

$$Temp = \frac{V_{25} - V_{sense}}{4.5} + 25$$

Table 14-2. *Temperature Sensor Characteristics for STM32F103x8 and STM32F103xB Devices*

Symbol	Parameter	Min	Typ	Max	Unit
TL	*Vsense* linearity with temperature	-	±1	±2	°C
Avg_Slope	Average slope	4.0	4.3	4.6	mV/°C
V_{25}	Voltage at 25 °C	1.34	1.43	1.52	V
tSTART	Startup time	4	-	10	µs
TS_temp	ADC sampling time when reading the temperature	-	-	17.1	µs

In Listing 14-3, you can see that the typical value of V_{25}=1.43 (x 100) was used from Table 14-2. The value 45 comes from 4.5 for Avg_Slope (the value x 10). This seemed to better match for the device I was using and is in the range listed. But if you find the computed value to be high, try reducing the value 45 to 43 (representing the slope of 4.3).

Another value of interest is T_{S_temp}, which is given as 17.1 µs. This is the sampling time performed by the testing that resulted in the tabled results. This is also the recommended sampling time found in RM0008.

When you don't need the temperature reading or the V_{ref}, you can save power consumption by turning them off, as follows:

```
adc_disable_temperature_sensor();
```

The datasheet also includes this note about temperature:

The temperature sensor output voltage changes linearly with temperature. The offset of this line varies from chip to chip due to process variation (up to 45 °C from one chip to another).

The internal temperature sensor is more suited to applications that detect temperature variations instead of absolute temperatures. If accurate temperature readings are needed, an external temperature sensor part should be used.

Voltage Reference

ADC1 allows you to read an internal voltage V_{ref}. This value can be used to calibrate the ADC to improve accuracy, with a typical value of 1.2 volts. Application Note AN2834 contains some very good information for those seeking the best available accuracy from the STM32 platform.

Analog Voltages

Reading 0 volts or +3.3 volts may not seem too exciting, so let's improve upon that. You can generate any voltage in that range with the help of a potentiometer. While a range of values from about 1 kohm to 15 kohms should be suitable, it is best to use the low end of this range for stable readings.

Figure 14-1 illustrates the 10-kohm potentiometer (or simply "pot") that I used for this experiment. If you're purchasing one, get a *linear* pot rather than an audio-tapered pot. Audio-tapered pots vary logarithmically to make volume controls change with the sense of hearing. North American suppliers will use a "B" prefix like "B10K" to indicate linear, like the one shown in Figure 14-1.
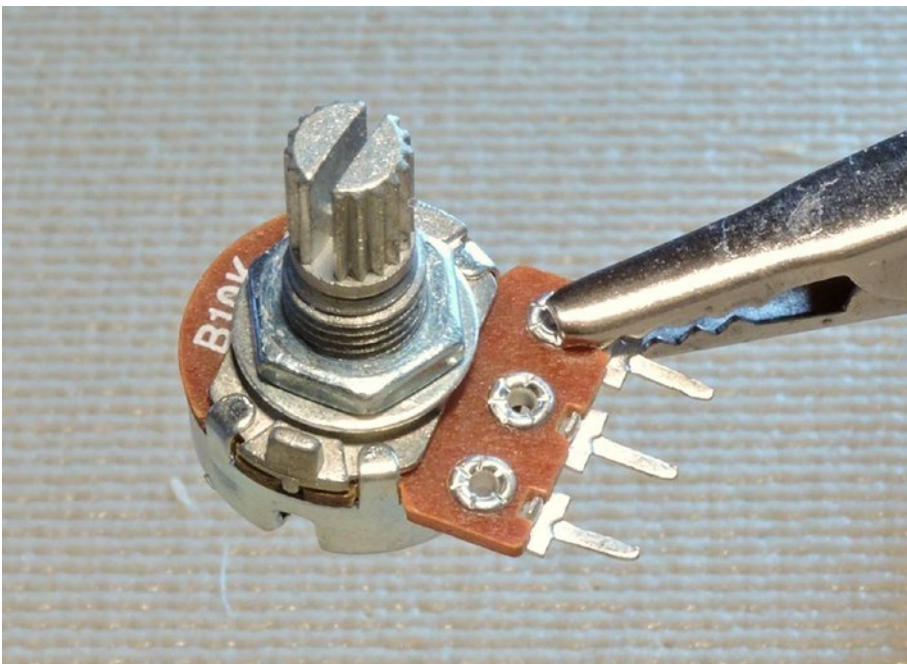
Figure 14-1. *A linear 10-kohm potentiometer*

The circuit is illustrated in Figure 14-2. You can use just one pot if you lack a second. Be sure to wire the lugs at opposite ends of the pot to the supply and ground. The center lug is connected to the wiper inside the pot and should be wired to your ADC input PA0 or PA1.

With minicom connected, you can turn the pots counter-clockwise. If the voltage reads near +3.3 volts when turned counter-clockwise, reverse the connections on the outer lugs of the pot. Corrected, it should read near zero. When you turn the pot midway, you should be able to read about +1.5 volts, and fully clockwise should return readings near +3.3 volts.
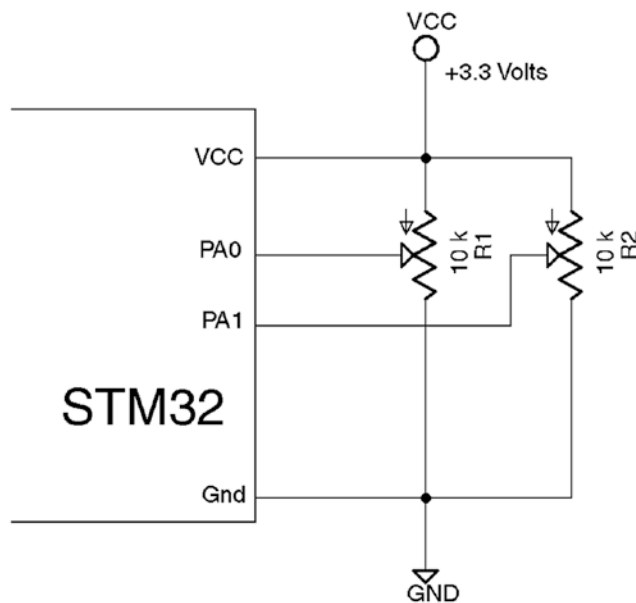
Figure 14-2. *A pair of potentiometers wired to ADC inputs PA0 and PA1*

Summary

The presented demo has just scratched the surface of what the STM32 ADC peripheral provides in the way of flexibility. In addition to single conversions, the ADC peripheral can be configured to use channel groups and perform scans. In addition to scanning any sequence of channels, it is also possible to have ADC values *injected* into the results. Finally, scanning and groups can include channels from peripherals ADC1 and ADC2.

This chapter gives you a simple place to start for ADC usage. Read chapter 11 of the STM32 reference manual RM0008 for the full extent of the ADC's sampling capabilities.[1]

EXERCISES

1. How is the internal STM32 temperature represented?

2. How does `GPIO_CNF_INPUT_ANALOG` differ from the values `GPIO_CNF_INPUT_PULL_UPDOWN` or `GPIO_CNF_INPUT_FLOAT`?

3. If PCLK has a frequency of 36 MHz, what would the ADC clock rate be when configured with a prescale divisor of 4?

4. Name three configuration options that affect the total power consumed by ADC.

5. Assuming that the ADC clock after the prescaler is 12 MHz, how long does the `ADC_SMPR_SMP_41DOT5CYC` configured sample take?

Bibliography

1. STMicroelectronics. Accessed January 12, 2018. `http://www.st.com/resource/en/reference_manual/cd00171190.pdf`

Clock Tree

Until this point, there's been an elephant in the room. We've configured and used various clocks without saying too much about them. Now is a good time to reveal some of the clock components that have been lurking in the shadows.

While asynchronous logic circuit designs exist, most microprocessors use one or more clocks. The STM32 series is no exception. This series is highly configurable, adding somewhat to its software complexity. But this added flexibility allows the designer to reduce power requirements by turning off peripherals and clocks that are not required.

This chapter will examine the clocks that the STM32F103C8T6 supports and how to configure them. This information will make it possible for you to calculate the correct prescaler counts needed to produce correct baud rates and SPI clock rates, and to correctly feed timers. It will also give you inside information needed to take advantage of special clock features and avoid pitfalls.

In the Beginning

Many clocks can be derived from others, but there has to be one or more sources at the start of any chain. Within the STM32F103C8T6 there are a total of *four* independent clock sources, as follows

1. 8 MHz RC oscillator (HSI)

2. 4–16 MHz crystal/ceramic oscillator (HSE)

3. 32.768 kHz crystal oscillator (LSE)

4. 40 kHz RC oscillator (LSI)

Table 15-1 summarizes the notation used for the preceding oscillators, as well as some of their major characteristics. For example, it is shown that the HSE oscillator is driven by a crystal and enjoys good stability, while the HSI oscillator is driven by RC (resistor and capacitor) and has relatively poor stability.

Table 15-1. *STM32 Oscillator Notation*

Notation	Low/High Speed	Internal/External	Driven By	Stability
LSI	Low speed	Internal	Resistor and Capacitor	Poor
LSE	Low speed	External	Crystal	Good
HSI	High speed	Internal	Resistor and Capacitor	Poor
HSE	High speed	External	Crystal	Good

RC Oscillators

A good question to ask is why provide RC oscillators if they are not that stable? For some applications, it may be enough that the MCU has a reasonable clock to execute instructions with. This saves the designer from having to supply a crystal and thus reduces the parts count.

Figure 15-1 illustrates the two crystals that are provided with the Blue Pill board. Notice the size of the 8.000 MHz crystal. Right beneath it in the photo is the 32.768 kHz crystal, which is housed in a rectangular blob of plastic. Relative to the MCU chip (above the 8 MHz crystal), these are large components.

The RC oscillator, as electronics folks know, consists of charging and discharging a capacitor through a resistor. The combination of capacitance and resistance determines the overall frequency. Creating capacitors inside of an IC (Integrated Circuit) presents challenges but is worth doing for chip buyers who want to reduce external components.

Note that the only STM32 RC oscillators provided are *internal* oscillators. Otherwise, resistors and capacitors would need to be supplied externally.
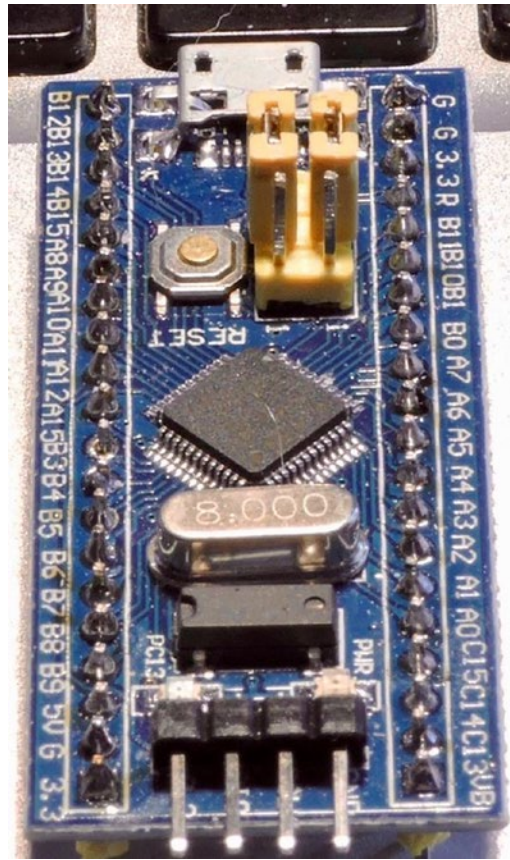
Figure 15-1. *8 MHz crystal and 32.768 kHz crystal below it*

Crystal Oscillators

Crystal oscillators are far more accurate and stable than the RC oscillator. However, they have the disadvantage that an *external* crystal must be supplied and wired up to the MCU chip. Figure 15-1 illustrates the two crystals found on the Blue Pill PCB, with the 8 MHz crystal used by the HSE oscillator. The 32.768 kHz crystal drives the LSE oscillator.

Oscillator Power

The higher rate at which the oscillator switches between low and high signal levels means additional current consumption. Every time the oscillator switches from low to high electrons have to be pushed into the circuit, thus requiring current flow (charging). When the oscillator switches from high to low, electrons have to be drawn out of the circuit and drained to ground (discharging). All of this requires energy.

It comes as no surprise then, if you do this charging and discharging more frequently in a given second, then more overall current is consumed. This is why so much attention is given to the clock design in the STM32 platform.

For some applications where the system is *battery* powered and the execution time is less important, it makes sense to use a lower-speed oscillator. If, on the other hand, the application is powered from a desktop over USB and speed is the dominant requirement, then higher oscillator rates are preferred.

Another selection criterion is accuracy. If you implement a serial link between different units, then you need to have an accurate notion of the baud rate. Having choice provides designers with different trade-offs.

Real-time Clock

The HSE, LSE, or LSI clock can be chosen for the source of the RTCCLK (real-time clock). Table 15-2 summarizes the clock configurations available. Note that the divisor is hardwired as 128 when the HSE clock is chosen.

Table 15-2. *Real-time Clock Sources When HSE Is 8 MHz*

Oscillator Source	Source Frequency	Divisor	Resulting Frequency
HSE	8.000 MHz	128	62.5 kHz
LSE	32.768 kHz	1	32.768 kHz
LSI	40 kHz	1	40 kHz

Watchdog Clock

The independent watchdog (IWDG) is hard wired to the LSI 40-kHz clock.

System Clock (SYSCLK)

The most interesting category of basic clock configuration is the system clock, from which other important clocks are derived. The SYSCLK can only be sourced from two of the four clock sources:

- HSI (RC), 8 MHz

- HSE (crystal), 4–16 MHz (8 MHz on Blue Pill)

There is one additional source that is derived from a phase-locked loop (PLL), which multiplies the frequency of the HSI or HSE clock input. When the source for the PLL is the HSI clock, the input is first divided by two. Table 15-3 provides a convenient table of values when HSI is used.

Table 15-3. *System Clock Derived from HSI and PLL*

Source	Frequency	PLL Multiplier	Resulting Frequency
HSI	8 MHz	No PLL	8 MHz
HSI	8 MHz ÷ 2	2	8 MHz
HSI	8 MHz ÷ 2	3	12 MHz
HSI	8 MHz ÷ 2	4	16 MHz
HSI	8 MHz ÷ 2	5	20 MHz
HSI	8 MHz ÷ 2	6	24 MHz
HSI	8 MHz ÷ 2	7	28 MHz
HSI	8 MHz ÷ 2	8	32 MHz
HSI	8 MHz ÷ 2	9	36 MHz
HSI	8 MHz ÷ 2	10	40 MHz
HSI	8 MHz ÷ 2	11	44 MHz
HSI	8 MHz ÷ 2	12	48 MHz
HSI	8 MHz ÷ 2	13	52 MHz
HSI	8 MHz ÷ 2	14	56 MHz
HSI	8 MHz ÷ 2	15	60 MHz
HSI	8 MHz ÷ 2	16	64 MHz

When the HSE is the chosen clock source, the calculation changes to the values shown in Table 15-4. The input to the PLL can use either the HSE divided by two (HSE ÷ 2) or not divided (HSE ÷ 1). The maximum usable system clock is 72 MHz.

Table 15-4. *System Clock Derived from HSE and PLL*

Source	Frequency	PLL Multiplier	HSE ÷ 2	HSE ÷ 1
HSE	8.000 MHz	No PLL	8 MHz	8 MHz
HSE	8.000 MHz	2	8 MHz	16 MHz
HSE	8.000 MHz	3	12 MHz	24 MHz
HSE	8.000 MHz	4	16 MHz	32 MHz
HSE	8.000 MHz	5	20 MHz	40 MHz
HSE	8.000 MHz	6	24 MHz	48 MHz
HSE	8.000 MHz	7	28 MHz	56 MHz
HSE	8.000 MHz	8	32 MHz	64 MHz
HSE	8.000 MHz	9	36 MHz	72 MHz
HSE	8.000 MHz	10	40 MHz	over limit
HSE	8.000 MHz	11	44 MHz	over limit
HSE	8.000 MHz	12	48 MHz	over limit
HSE	8.000 MHz	13	52 MHz	over limit
HSE	8.000 MHz	14	56 MHz	over limit
HSE	8.000 MHz	15	60 MHz	over limit
HSE	8.000 MHz	16	64 MHz	over limit

Figure 15-2 provides a slightly simplified diagram of the system clock tree up to the point of the SYSCLK. The asterisks identify what is normally configured for the Blue Pill STM32F103C8T6.
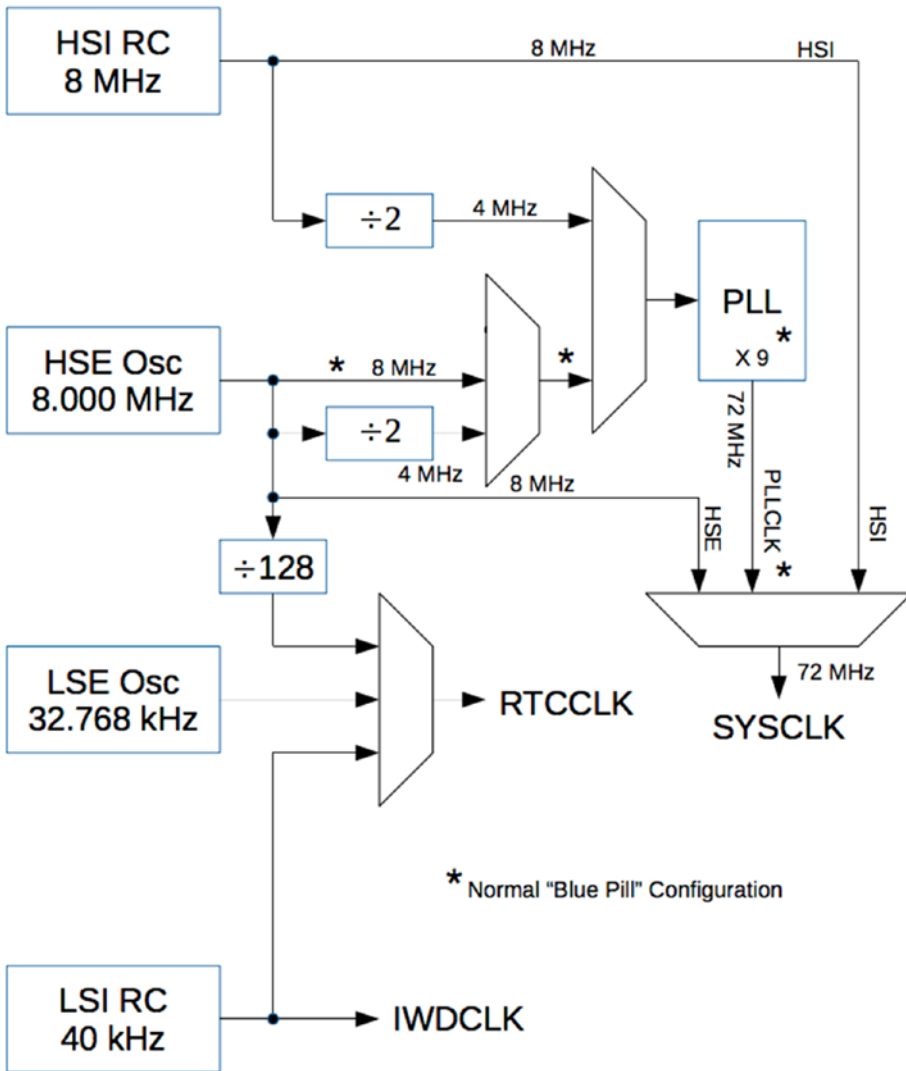
Figure 15-2. *Simplified summary of clock tree up to the point of SYSCLK*

SYSCLK and USB

If you're not using USB, you can ignore this issue. But when USB support is required, your choices are limited, as outlined in Table 15-5. The USB prescaler must be set so that the USBCLK is 48 MHz.

Table 15-5. *Valid Clock Configurations when USB Is Used*

SYSCLK Frequency	USB Divisor	Resulting USB Clock
72 MHz	÷ 1.5	48 MHz
48 MHz	÷ 1	48 MHz

AHB Bus

Throughout the ST Microelectronics document RM0008, which describes the STM32 series, references to AHB are made without ever explaining what it is. So, what is the AHB anyway? Wikipedia helps with this:[1]

> *The ARM Advanced Microcontroller Bus Architecture (AMBA) is an open-standard, on-chip interconnect specification for the connection and management of functional blocks in system-on-a-chip (SoC) designs. . . . AMBA was introduced by ARM in 1996. The first AMBA buses were Advanced System Bus (ASB) and Advanced Peripheral Bus (APB). In its second version, AMBA 2 in 1999, ARM added AMBA High-performance Bus (AHB) that is a single clock-edge protocol.*

There we have it—AHB is the AMBA high-performance bus. Within the STM32 family, the AHB has a prescaler that uses the SYSCLK as the input. Assuming that the SYSCLK is 72 MHz, Table 15-6 summarizes the AHB choices.

Table 15-6. *STM32F103C8T6 AHB Frequencies with a 72 MHz SYSCLK*

Bit Value	Divisor	Resulting Frequency
0xxx	SYSCLK not divided	72 MHz
1000	SYSCLK divided by 2	36 MHz
1001	SYSCLK divided by 4	18 MHz
1010	SYSCLK divided by 8	9 MHz
1011	SYSCLK divided by 16	4.5 MHz
1100	SYSCLK divided by 64	1.125 MHz
1101	SYSCLK divided by 128	562.5 kHz
1110	SYSCLK divided by 256	281.25 kHz
1111	SYSCLK divided by 512	140.625 kHz

Starting from SYSCLK, Figure 15-3 illustrates why this is called a clock tree. From the SYSCLK signal, many other clocks are derived from configured divisors and enables.
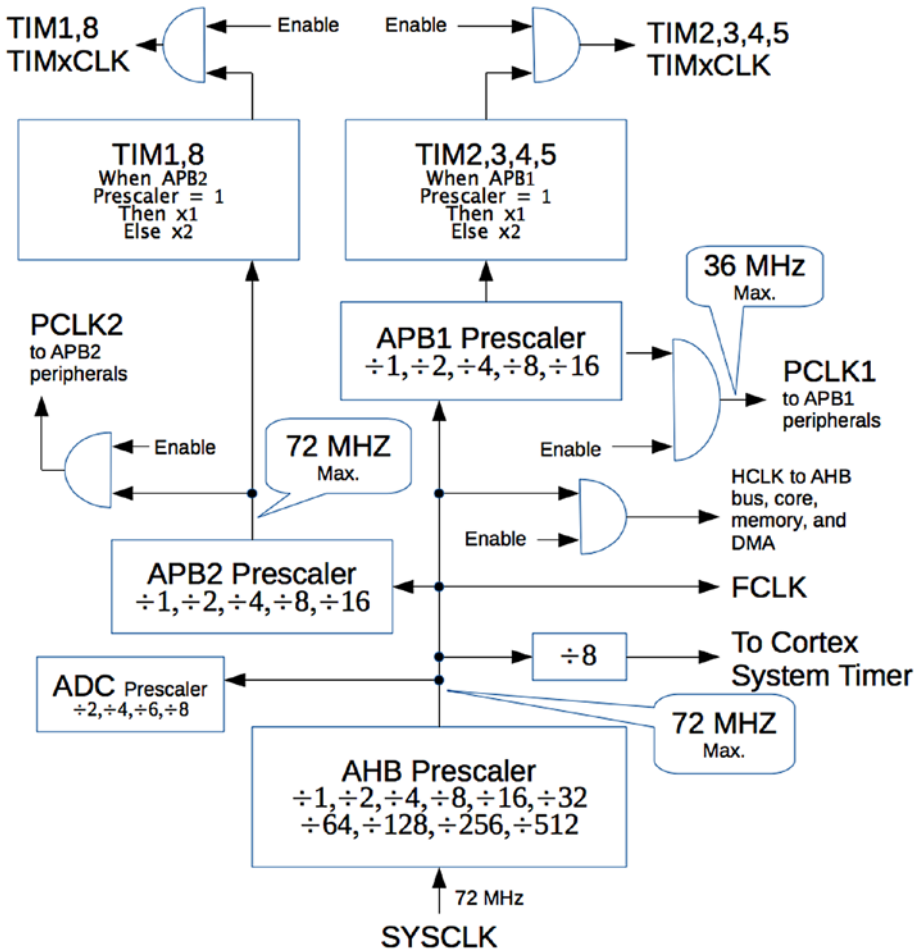
Figure 15-3. *Clock tree starting from SYSCLK*

rcc_clock_setup_in_hse_8mhz_out_72mhz( )

In most of the demos presented in this book, the following libopencm3 routine is used at the start of the main program:

```
rcc_clock_setup_in_hse_8mhz_out_72mhz();
```

To help us understand specifically what that is doing, Listing 15-1 illustrates the libopencm3 function code for it.

Listing 15-1. The Function rcc_clock_setup_in_hse_8mhz_out_72mhz( )

```
0911: void rcc_clock_setup_in_hse_8mhz_out_72mhz(void)
0912: {
0913:   /* Enable internal high-speed oscillator. */
0914:   rcc_osc_on(RCC_HSI);
0915:   rcc_wait_for_osc_ready(RCC_HSI);
0916:
0917:   /* Select HSI as SYSCLK source. */
0918:   rcc_set_sysclk_source(RCC_CFGR_SW_SYSCLKSEL_HSICLK);
0919:
0920:   /* Enable external high-speed oscillator 8MHz. */
0921:   rcc_osc_on(RCC_HSE);
0922:   rcc_wait_for_osc_ready(RCC_HSE);
0923:   rcc_set_sysclk_source(RCC_CFGR_SW_SYSCLKSEL_HSECLK);
0924:
0925:   /*
0926:    * Set prescalers for AHB, ADC, ABP1, ABP2.
0927:    * Do this before touching the PLL (TODO: why?).
0928:    */
0929:   rcc_set_hpre(RCC_CFGR_HPRE_SYSCLK_NODIV);/* Set. 72MHz Max. 72MHz */
0930:   rcc_set_adcpre(RCC_CFGR_ADCPRE_PCLK2_DIV8);/*Set. 9MHz Max. 14MHz */
0931:   rcc_set_ppre1(RCC_CFGR_PPRE1_HCLK_DIV2); /* Set. 36MHz Max. 36MHz */
0932:   rcc_set_ppre2(RCC_CFGR_PPRE2_HCLK_NODIV);/* Set. 72MHz Max. 72MHz */
0933:
0934:   /*
0935:    * Sysclk runs with 72MHz -> 2 waitstates.
0936:    * 0WS from 0-24MHz
0937:    * 1WS from 24-48MHz
0938:    * 2WS from 48-72MHz
0939:    */
0940:   flash_set_ws(FLASH_ACR_LATENCY_2WS);
0941:
```

```
0942:    /*
0943:     * Set the PLL multiplication factor to 9.
0944:     * 8MHz (external) * 9 (multiplier) = 72MHz
0945:     */
0946:    rcc_set_pll_multiplication_factor(RCC_CFGR_PLLMUL_PLL_CLK_MUL9);
0947:
0948:    /* Select HSE as PLL source. */
0949:    rcc_set_pll_source(RCC_CFGR_PLLSRC_HSE_CLK);
0950:
0951:    /*
0952:     * External frequency undivided before entering PLL
0953:     * (only valid/needed for HSE).
0954:     */
0955:    rcc_set_pllxtpre(RCC_CFGR_PLLXTPRE_HSE_CLK);
0956:
0957:    /* Enable PLL oscillator and wait for it to stabilize. */
0958:    rcc_osc_on(RCC_PLL);
0959:    rcc_wait_for_osc_ready(RCC_PLL);
0960:
0961:    /* Select PLL as SYSCLK source. */
0962:    rcc_set_sysclk_source(RCC_CFGR_SW_SYSCLKSEL_PLLCLK);
0963:
0964:    /* Set the peripheral clock frequencies used */
0965:    rcc_ahb_frequency = 72000000;
0966:    rcc_apb1_frequency = 36000000;
0967:    rcc_apb2_frequency = 72000000;
0968: }
```

The basic steps used are the following:

1. The HSI oscillator is turned on and waits for it to become ready (lines 914 to 915).

2. Selects the HSI oscillator as the SYSCLK source (line 918).

3. The HSE (8 MHz crystal oscillator) is enabled in line 921, and the code waits until it is ready (line 922).

4. The SYSCLK is then switched to use the HSE clock (line 923). Note that this is not the PLL yet, so at this point the SYSCLK is 8.000 MHz as determined by the crystal.

5. The AHB is set to use no divisor in its prescaler (line 929), resulting in an input AHB clock of 72 MHz after the PLL is selected (later) as the clock source.

6. The ADC prescaler is configured with a divisor of 8 (line 930), which results in a frequency of 9 MHz (after switch to the PLL). As the comment indicates, it must not exceed 14 MHz.

7. The prescaler for APB1 (Advanced Peripheral Bus 1) is set to divide by 2, resulting in an APB1 clock of 36 MHz after switch to the PLL (line 931). This is the maximum frequency for this bus.

8. The prescaler for APB2 is set to use no divisor, resulting in an APB2 frequency of 72 MHz when switched later to use the PLL (line 932). This is also the maximum frequency for APB2.

9. Since the SYSCLK runs at 72 MHz, there must be two wait cycles inserted for each flash memory access (line 940).

10. The PLL is now set with a multiplier of 9 to set its output clock to 72 MHz (line 946).

11. Line 955 removes any ÷ 2 setting for HSE entering the PLL that might be set.

12. Finally, line 962 selects the PLL as the SYSCLK source. This increases the SYSCLK from 8 to 72 MHz, with the AHB bus now operating at 72 MHz, APB1 running at 36 MHz, and APB2 running at 72 MHz.

13. Lines 965 to 967 set global values `rcc_ahb_frequency`, `rcc_apb1_frequency` and `rcc_apb2_frequency` for application use.

The globals are defined in `rcc.h` and are defined as follows:

```
#include <libopencm3/stm32/rcc.h>

extern uint32_t rcc_ahb_frequency;
extern uint32_t rcc_apb1_frequency;
extern uint32_t rcc_apb2_frequency;
```

From this, you can see that there is quite a bit that must be done at startup to make sure that no clocks falter or fail.

APB1 Peripherals

Each peripheral connected to the APB1 bus in the Blue Pill device receives a 36 MHz clock (unless otherwise configured). Each peripheral, however, has a private AND-gate to enable/disable this clock in order to save power. To enable the receipt of the clock, the peripheral must enable it. For example, the CAN peripheral must enable the clock for the peripheral itself. The same applies to APB1 timer peripherals.

APB2 Peripherals

Like the APB1 peripherals, each peripheral attached to the APB2 bus must enable/disable the receipt of their own 72 MHz clock. This also applies to APB2 timer peripherals.

Timers

Special mention is made of timers here because there is a not-so-obvious wrinkle in their configuration. APB1 and APB2 timers have a prescaler, allowing their bus clocks to be divided down for a lower frequency. The *exception*, however, is that when the APB1/APB2 prescaler is set to one, the bus frequency is *multiplied by two!*

Note   When a timer prescaler is set to 1, the output of the prescaler is a bus frequency times 2!

---

rcc_set_mco( )

The libopencm3 library provides a function named rcc_set_mco() to configure a clock output on GPIO PA8. The valid macro values passed as an argument are described in Table .

```
rcc_set_mco(macro);
```

Table 15-7. *Valid Arguments to rcc_set_mco( )*

Macro Name	Value	Description
RCC_CFGR_MCO_NOCLK	0 x 0	No clock to MCO (disconnected)
RCC_CFGR_MCO_SYSCLK	0 x 4	SYSCLK to MCO
RCC_CFGR_MCO_HSI	0 x 5	HSI to MCO
RCC_CFGR_MCO_HSE	0 x 6	HSE to MCO
RCC_CFGR_MCO_PLL_DIV2	0 x 7	PLL $\div$ 2 to MCO

Calling the routine rcc_set_mco() by itself is not enough. The GPIO PA8 must be configured for *alternate function I/O*:

```
rcc_periph_clock_enable(RCC_GPIOA);
gpio_set_mode(GPIOA,
    GPIO_MODE_OUTPUT_50_MHZ,
    GPIO_CNF_OUTPUT_ALTFN_PUSHPULL, // ALTFN
    GPIO8);                         // PA8=MCO
```

HSI Demo

The example code for the HSI clock demo is located here:

```
$ cd ~/stm32f103c8t6/hsi
$ make clobber
$ make
$ make flash
```

Note that this demo does not use FreeRTOS. Its code is very basic and simply arranges to have the HSI clock brought out to GPIO pin PA8. Listing 15-2 shows the main program responsible.

Listing 15-2. The hsi.c Demonstration Program

```
0010: int
0011: main(void) {
0012:
0013:    // LED Configuration:
0014:    rcc_periph_clock_enable(RCC_GPIOC);
0015:    gpio_set_mode(GPIOC,GPIO_MODE_OUTPUT_2_MHZ,
0016:             GPIO_CNF_OUTPUT_PUSHPULL,GPIO13);
0017:    gpio_clear(GPIOC,GPIO13);    // LED Off
0018:
0019:    // MCO Configuration:
0020:    rcc_periph_clock_enable(RCC_GPIOA);
0021:    gpio_set_mode(GPIOA,
0022:        GPIO_MODE_OUTPUT_50_MHZ,
0023:        GPIO_CNF_OUTPUT_ALTFN_PUSHPULL,
0024:        GPIO8);              // PA8=MCO
0025:
0026:    rcc_set_mco(RCC_CFGR_MCO_HSI);
0027:
0028:    gpio_set(GPIOC,GPIO13); // LED On
0029:    for (;;);
0030:    return 0;
0031: }
```

Aside from configuring the LED PC13, the main elements are as follows:

1. The GPIOA peripheral clock is enabled in line 20.

2. The GPIOA to which pin PA8 is configured for output (max 50 MHz, line 22) is an alternate function (line 23) in push/pull mode.

3. The HSI clock is directed to PA8 in line 26.

After flashing the STM32, you should be able to see the HSI clock on PA8 with a scope or DSO as soon as power is applied or after reset (Figure 15-4). From the figure, you can see that the HSI clock is near 8 MHz.

Figure 15-4. *The HSI MCO trace*

HSE Demo

The example code for the HSE clock demo is located here:

```
$ cd ~/stm32f103c8t6/hse
$ make clobber
$ make
$ make flash
```

Note that this demo also does not use FreeRTOS. Its code is basic and simply arranges to have the HSE clock brought out to GPIO pin PA8. The only difference between this demo program and the HSI demo is one line:

```
rcc_set_mco(RCC_CFGR_MCO_HSE);
```

After flashing the STM32, you should be able to see the HSE clock on PA8 with a scope or DSO as soon as power is applied (Figure 15-5). In the figure, the frequency is more accurate to 8 MHz.

Figure 15-5. The HSE MCO trace (note how similar this is to Figure 15-4)

PLL ÷ 2 Demo

The example code for the PLL ÷ 2 clock demo is located here:

```
$ cd ~/stm32f103c8t6/mco_pll2
$ make clobber
$ make
$ make flash
```

Note again that this demo also does not use FreeRTOS. Its code is basic and simply arranges to have the PLL ÷ 2 clock brought out to GPIO pin PA8. The only difference between this demo program and the HSE demo is one line:

```
rcc_set_mco(RCC_CFGR_MCO_PLL_DIV2);
```

Having the PLL ÷ 2 clock sent out to PA8 is helpful because the GPIO pin is limited to driving 50 MHz. You could attempt to send 72 MHz out, but the signal would be badly degraded and perhaps stress the active components involved. But 36 MHz is well within the acceptable performance range.

After flashing the STM32, you should be able to see the PLL ÷ 2 clock on PA8 with a scope or DSO as soon as power is applied. Notice that the frequency shown is near 36 MHz, as expected (72 MHz ÷ 2), in Figure 15-6.
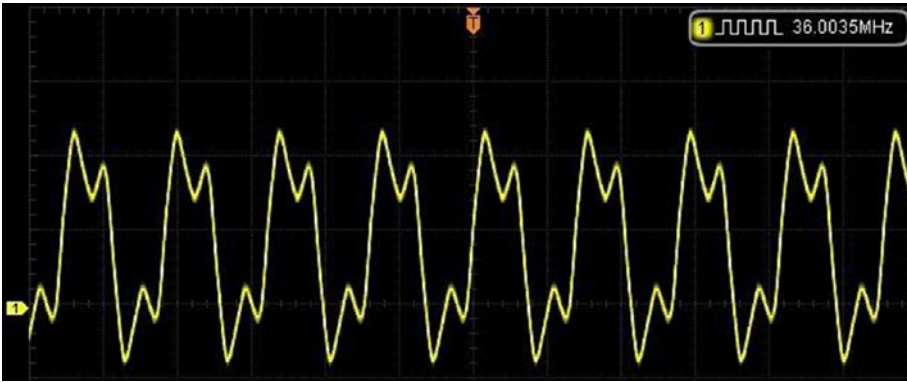
Figure 15-6. *The PLL ÷ 2 MCO trace*

Summary

In this chapter, the overview of the clock-tree system was presented, starting with the main clock sources: HSI and HSE for the *system* clock, and HSE, LSE, and LSI for the *real-time* clock. Clock LSI was also used by the watchdog timer.

The next main category of clocks stem *from* the *system* clock. The system clock is able to employ the use of the *PLL*, which is capable of multiplying its input clock up to 72 MHz. From the system clock, an *AHB* clock is derived. Then, from the AHB clock are derived APB1 and APB2 clocks.

Finally, it was noted that each peripheral needing a clock has its own gate that it must enable in order to use a given clock. This design saves power by leaving unneeded clocks disabled.

The HSI, HSE, and PLL ÷ 2 demos illustrated how to check a clock that is otherwise internal and unseen. It is also possible that a clock placed on PA8 may have its application to external peripherals needing an input clock.

Bibliography

1. "Advanced Microcontroller Bus Architecture." Wikipedia. December 16, 2017. Accessed December 29, 2017. https:// en.wikipedia.org/wiki/Advanced_Microcontroller_Bus_ Architecture

EXERCISES

1.   What is the advantage of an RC clock?

2.   What is the disadvantage of an RC clock?

3.   What is the advantage of a crystal-derived clock?

4.   What is the PLL used for?

5.   What does AHB stand for?

6.   Why must the GPIO PA8 be configured with `GPIO_CNF_OUTPUT_ALTFN_PUSHPULL`?

PWM with Timer 2

The STM32 family has a complex array of options available for operating timers. The STM32F103 has general-purpose timers TIM2, 3, 4, and 5 and then advanced timers TIM1 and TIM8. The general-purpose timers have plenty of features so you won't need to reach for the advanced ones.

This chapter will demonstrate one of the frequently sought-after applications of a timer–that of driving a PWM (Pulse Width Modulated) servo motor. Figure 16-1 illustrates one example of a typical servo motor, which was pulled out of service.
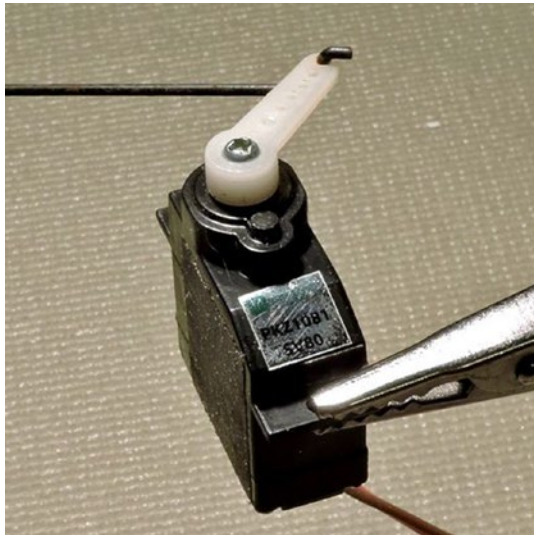
Figure 16-1. *A typical RC (Radio Controlled) servo motor (PKZ1081 SV80)*

PWM Signals

What does a PWM signal look like? Figure 16-2 illustrates one. The sample shown has a high pulse that is 2.6 ms long (note the cursors and the BX-AX value shown). The measured frequency was about 147 Hz. This means the entire period of the signal is about 6.8 ms.

Most RC controls are based upon the length of time that the signal is high rather than the duty cycle, but control systems do vary.
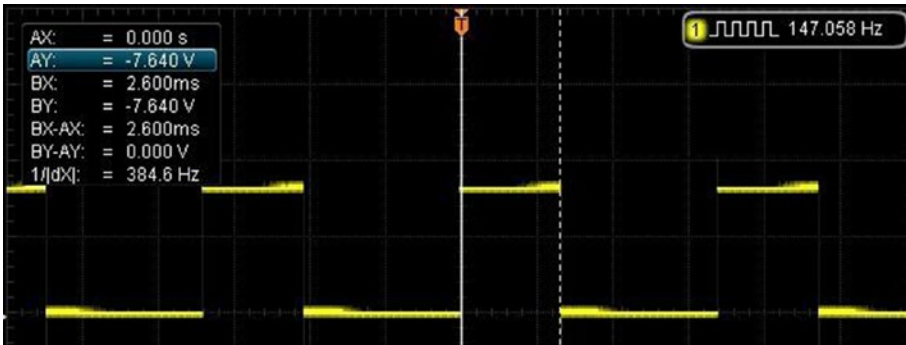
Figure 16-2. *A PWM signal with a period of 147 Hz and pulse width of 2.6 ms*

Timer 2

Timers 2 to 5 are general-purpose timers in the STM32F103C8T6. Despite being general purpose, they are quite flexible. Their overall features include the following:

- 16-bit up, down, up/down auto-reload counter

- 16-bit prescaler to divide the counter-clock frequency by 1 to 65536

- Prescaler can be changed "on the fly."

- Up to four independent channels for:

  - input capture

  - output capture

  - PWM generation (edge- and center-aligned modes)

  - One-pulse mode output

- Synchronization circuit controlling timer with external signals and for interconnection with other timers

- Interrupt/DMA generation:

  - Update counter overflow/underflow, counter initialization by software or trigger

- Trigger event (counter start, stop, initialization, or count by internal/external trigger)

- Input capture

- Output capture

- Trigger input

Section 15 of the RM0008[1] reference manual discusses all of this, but in this chapter we'll focus on the generation of a PWM signal. The software for this chapter's demonstration is found in the following directory:

```
$ cd ~/stm32f103c8t6/rtos/tim2_pwm
```

The source code is entirely in the file main.c. The portions of task1() that apply to the timer will be listed in small sections to ease the discussion. Some initial configuration is shown in Listing 16-1.

Listing 16-1. Configuration of PA1 for Timer 2 Output

```
0029:   rcc_periph_clock_enable(RCC_TIM2);
0030:   rcc_periph_clock_enable(RCC_AFIO);
0031:
0032:   // PA1 == TIM2.CH2
0033:   rcc_periph_clock_enable(RCC_GPIOA);
0034:   gpio_primary_remap(
0035:       AFIO_MAPR_SWJ_CFG_JTAG_OFF_SW_OFF,  // Optional
0036:       AFIO_MAPR_TIM2_REMAP_NO_REMAP);     // default: TIM2.CH2=GPIOA1
0037:   gpio_set_mode(GPIOA,GPIO_MODE_OUTPUT_50_MHZ,   // High speed
0038:       GPIO_CNF_OUTPUT_ALTFN_PUSHPULL,GPIO1); // GPIOA1=TIM2.CH2
```

Line 29 enables the clock for Timer 2, while AFIO's clock is enabled in line 30. This needs to be done before line 35.

Line 33 enables the clock for GPIOA for PA1's use. Lines 34 to 36 are an AFIO call that says use the default mapping where channel 2 of Timer 2 comes out on PA1 (this can be omitted in the default case). Change this call if you need it to come out to PB3 instead. Finally, lines 37 and 38 use the ALTFN macro to connect the GPIO pin to the timer for PA1. This is critical.

Listing 16-2 illustrates code that initializes the timer and establishes its operating mode.

Listing 16-2. Initialize Timer 2 and Set Its Mode

```
0042:    // TIM2:
0043:    timer_disable_counter(TIM2);
0044:    timer_reset(TIM2);
0045:
0046:    timer_set_mode(TIM2,
0047:        TIM_CR1_CKD_CK_INT,
0048:        TIM_CR1_CMS_EDGE,
0049:        TIM_CR1_DIR_UP);
```

Lines 43 and 44 disable the counter and reset Timer 2. Line 46 then establishes the operating mode for the timer as follows:

- `TIM_CR1_CKD_CK_INT` configures the division ration between the timer clock (`CLK_INT`) frequency and the sampling clock used by the digital filters. Here, we don't use the digital filters, so this macro sets the digital filter frequency equal to the clock frequency (see datasheet `TIMx_CR1.CKD` for Timer 2 for more).

- `TIM_CR1_CMS_EDGE` specifies that the edge-aligned mode is to be used (versus center-aligned).

- `TIM_CR1_DIR_UP` specifies that the counter will count *up*.

Listing 16-3 continues with the Timer 2 configuration and then launches into the demonstration.

Listing 16-3. Remainder of Configuration and Timer Launch

```
0026:    static const int ms[4] = { 500, 1200, 2500, 1200 };
0027:    int msx = 0;
...
0050:    timer_set_prescaler(TIM2,72);
0051:    // Only needed for advanced timers:
0052:    // timer_set_repetition_counter(TIM2,0);
0053:    timer_enable_preload(TIM2);
0054:    timer_continuous_mode(TIM2);
0055:    timer_set_period(TIM2,33333);
```

```
0056:
0057:    timer_disable_oc_output(TIM2,TIM_OC2);
0058:    timer_set_oc_mode(TIM2,TIM_OC2,TIM_OCM_PWM1);
0059:    timer_enable_oc_output(TIM2,TIM_OC2);
0060:
0061:    timer_set_oc_value(TIM2,TIM_OC2,ms[msx=0]);
0062:    timer_enable_counter(TIM2);
```

Line 50 establishes the timer frequency by setting the prescaler for it. This will be described fully later. Line 52 is needed for the advanced timers only (Timers 1 and 8) and is thus commented out. This call is ignored by libopencm3 for Timers 2 through 5.

Lines 53 and 54 configure two more options for the timer. The timer_enable_ preload() call indicates that the TIM2_ARR register is buffered (for reloading). The function timer_continuous_mode() configures the timer to keep running rather than to stop after one pulse. Line 55 sets the maximum timer count to establish its period. More will be said about this later.

Lines 57 to 59 configure Timer 2's channel OC2 (output-compare channel 2) to operate in PWM1 mode. The configuration to PWM1 mode occurs in line 58. TIM_OCM_ PWM1 specifies the following:

When counting up, the output channel is active (high) when the timer's count is less than the timer capture/compare register, or else the channel goes low.

Line 61 sets the output-compare register to the value found in the ms[] array. This establishes a starting pulse width (in microseconds). The timer is finally started in line 62.

Line 26 declares an array ms[4] containing four values. These are pulse widths in microseconds, with 1200 (1.2 ms) as the center position. With the mode established in line 58, the following will happen:

- Counter values 0 through ms[msx]-1 will cause GPIO PA1 to go high (while it is considered active).

- Once the counter climbs above that value, the PA1 level goes low.

With this configuration, the PA1 output will initially go high for 500 µsec (0.5 ms) for a total of 33,333 counts (the period configured in line 55).

PWM Loop

The demonstration program performs the loop illustrated in Listing 16-4.

Listing 16-4. Demonstration PWM Loop

```
0064:   for (;;) {
0065:       vTaskDelay(1000);
0066:       gpio_toggle(GPIOC,GPIO13);
0067:       msx = (msx+1) % 4;
0068:       timer_set_oc_value(TIM2,TIM_OC2,ms[msx]);
0069:   }
```

At the top of the loop, line 65 delays execution for about one second (1000 ms), after which the PC13 LED GPIO is toggled (line 66). Line 67 updates array index variable msx so that it counts up, but starts over at zero if it goes past three. Using the index variable, the next position value is used to change the output-compare register in line 68. Once the servo sees this pulse, its position will change (or you can view the pulse-width change on the scope).

Calculating Timer Prescale

In Listing 16-3, line 50 was a function call that established the prescaler count. Let's break that calculation down. For your convenience, the prescaler setting was this:

```
timer_set_prescaler(TIM2,72)
```

The input to the counter is 72 MHz because when the Blue Pill is configured the APB1 prescaler is normally set to 2, and thus the bus frequency is divided down to 36 MHz (its maximum). What does that note about the TIM2, 3, 4, and 5 prescaler say?

When APB1 prescaler = 1, then is times 1, else it is times 2.

You could be excused if you got confused by all of this. After all, we have the 72 MHz SYSCLCK frequency divided down by 2 to meet the 36 MHz maximum frequency for the APB1 bus. After that, there is another prescaler that applies for timers 2 through 5. I'll refer to this as the *global* prescaler since it applies to all timers (2 through 5). The output of that prescaler feeds into the timers' own *private* prescalers. With all these prescalers, it's no wonder there is confusion!

The quote about when prescaler = 1 reminds us of the fact that what comes out of the *global* timer prescaler is actually the APB1 bus frequency *times two!* Therefore, what goes into Timer 2's own *private* prescaler is 72 MHz, not 36. So now we can explain the top part of the formula:

$$\frac{72000000}{72} = 1000000$$

The numerator represents the 72 MHz entering Timer 2's *private* prescaler. Supplying a private timer prescale value of 72 causes the timer to be updated at 1 MHz (line 50 of Listing 16-3).

We didn't have to use this ratio, but it proves to be convenient. Each count occurs in 1 μsec, allowing us to specify the pulse width in *microseconds.*

30 Hz Cycle

I have assumed that your RC servo needs a cycle rate of 30 Hz. This is defined by the configuration performed in Listing 16-3 line 55:

$$\frac{\frac{f_{APB1} \times 2}{prescaler}}{f_{period}} = \frac{\frac{36000000 \times 2}{72}}{30} = 33333.3$$

To program it, we could code:

```
timer_set_prescaler(TIM2,36000000*2/72/30);
```

or simply code:

```
timer_set_prescaler(TIM2,33333);
```

To reduce the period (increase the frequency) to 50 Hz, simply replace 30 with 50 in the calculation.

Tip   Remember that the frequency entering the timer's private prescaler is doubled if the APBx prescaler is 1.

Servo Hookup

Unfortunately, servos generally operate at around 6 volts. For the STM32, this requires a small driver circuit to bridge the gap.

The good news is that the interface is rather simple. You can use a CD4050BE CMOS IC to accept a 3.3-volt signal on its input and produce a nearly 6-volt level on its output (Figure 16-3). Notice that pin 1 of IC1A is connected to the servo motor's supply. The design of the CD4050 is such that the input (IC1A pin 3) can be safely connected to the STM32.
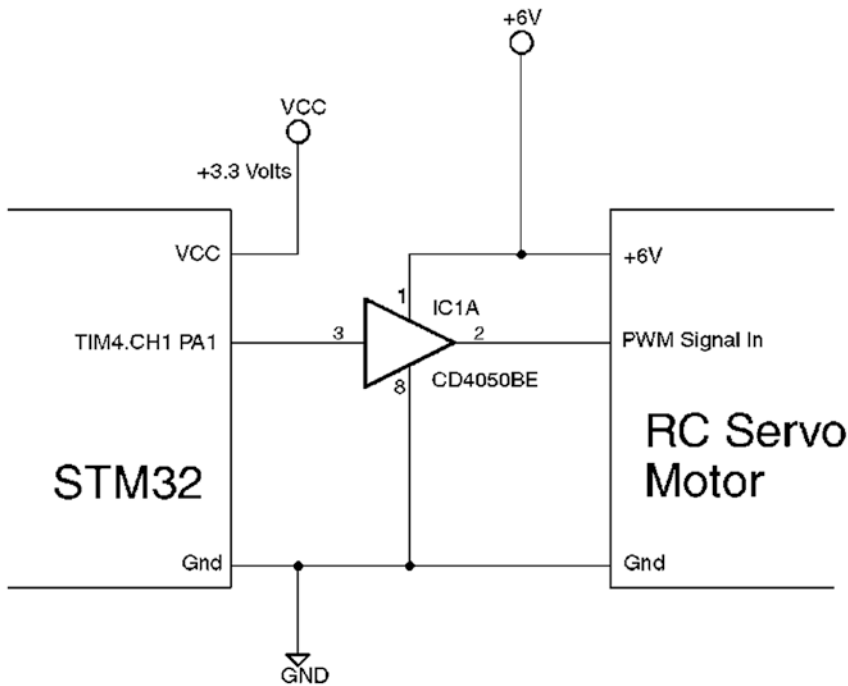
Figure 16-3. *The 6-volt interface circuit between STM32 MCU and servo motor*

Other replacements for the CD4050 would be the 74HCT244 or 74HCT245 (with different pinouts). It is critical that these special-talent chips are used to bridge the voltage gap. For more about this, see the book *Custom Raspberry Pi Interfaces*, Chapter 2, "3V/5V Signal Interfacing." While other CMOS chips can operate at 6 volts, they may not see the STM32 input signal as a high (this depends upon the V_{IH} threshold value).

When hooking up your circuit, make certain that you connect the 6-volt system ground to the STM32 ground. This provides a common voltage reference point.

Running the Demo

After building and flashing the software as follows:

```
$ make clobber
$ make
$ make flash
```

all you have to do is make sure your connections are correct and plug in the power (or power the STM32 from USB). *No USB or UART communication is used by this demo.*

With the servo connected, it should twitch every second to one extreme, middle position, other extreme, middle again, and back to the first extreme. Servos vary in their PWM requirements, so you may need to change the following:

- The pulse-width table in Listing 16-3, line 26 (these are in microseconds)

- The period in Listing 16-3, line 55

For amusement, attach a cat's laser pointer to the servo arm.

PWM on PB3

Timer 2 output-compare 2 can be redirected to PB3. This can be exploited if you require a 5-volt PWM signal. PB3 is a 5-volt-tolerant GPIO, though it can't produce a 5-volt high signal directly. When driven as an *open-drain* GPIO, however, a pull-up resistor can make the signal rise to 5 volts.

The source code for this version of the project is located here:

```
$ cd stm32f103c8t6/rtos/tim2_pwm_pb3
```

The `main.c` module is nearly identical except for the differences shown here:

```
$ diff -c ../tim2_pwm/main.c main.c
...
!     // PA1 == TIM2.CH2
!     rcc_periph_clock_enable(RCC_GPIOA);      // Need GPIOA clock
      gpio_primary_remap(
          AFIO_MAPR_SWJ_CFG_JTAG_OFF_SW_OFF, // Optional
```

```
!          AFIO_MAPR_TIM2_REMAP_NO_REMAP);     // default: TIM2.CH2=GPIOA1
!     gpio_set_mode(GPIOA,GPIO_MODE_OUTPUT_50_MHZ, // High speed
!          GPIO_CNF_OUTPUT_ALTFN_PUSHPULL,GPIO1);   // GPIOA1=TIM2.CH2
--- 29,41 ----
!     // PB3 == TIM2.CH2
!     rcc_periph_clock_enable(RCC_GPIOB);          // Need GPIOB clock
      gpio_primary_remap(
          AFIO_MAPR_SWJ_CFG_JTAG_OFF_SW_OFF,       // Optional
!          AFIO_MAPR_TIM2_REMAP_PARTIAL_REMAP1);    // TIM2.CH2=PB3
!     gpio_set_mode(GPIOB,GPIO_MODE_OUTPUT_50_MHZ, // High speed
!          GPIO_CNF_OUTPUT_ALTFN_OPENDRAIN,GPIO3);  // PB3=TIM2.CH2
```

The first change is to activate GPIOB instead of GPIOA. Following that, the gpio_primary_remap() call uses argument AFIO_MAPR_TIM2_REMAP_PARTIAL_REMAP1 to direct Timer 2's output-compare 2 to PB3.

The last change in gpio_set_mode() configures PB3 to use *open-drain* output. This is necessary because PB3 cannot *produce* a 5-volt signal directly (it can only pull it up to +3.3 volts). PB3 can, however, allow it to be *pulled up* to +5 volts by a resistor when it is operating as open drain. This change and the addition of a pullup resistor in the range of 2K to 10K ohms will permit a 5-volt output signal to be generated.

Other Timers

When it comes to servo PWM signals, people often want to know how many PWM channels can be made available. Table 16-1 summarizes the timers available on the STM32F103C8T6 and the default GPIO assignments. Some timers also have alternate assignments available.

Table 16-1. *Timers Available to the STM32F103C8T6*

Timer	Type	Channel 1	Channel 2	Channel 3	Channel 4
TIM1	Advanced	PA12	PA8	PA9	PA10
		PB13=CH1N	PB14=CH2N	PB15=CH3N	PB12=BKIN
TIM2	General Purpose	PA0	PA1	PA2	PA3
TIM3	General Purpose	PA6	PA7	PB0	PB1
TIM4	General Purpose	PB6	PB7	PB8	PB9
TIM5	General Purpose	None	None	None	PA3
TIM8	Advanced	None	None	None	None

Timers have four channels, which are configurable as GPIO inputs or outputs. Timer TIM8 has no GPIO connections at all but can link internally to other timers. TIM5 only links its channel 4 to PA3. The remaining general-purpose timers TIM2 through TIM4 have a full complement of GPIOs.

Advanced timer TIM1 has the most comprehensive set of I/Os, with up to eight assignments. Entries marked with CHxN are GPIOs that use the opposite polarity. Finally, the signal BKIN serves as a special "break" input.

The answer to the question "How many PWM *timers*?" is five. To use all five, you have to accept GPIO PA3 for TIM5. TIM1 through TIM4 can produce PWM output signals on four GPIO-connected channels. Altogether, these five timers provide a possible total of twenty-one output *channels*.

More PWM Channels

Getting more PWM output channels requires a little organization and software. Each timer has four channels, so TIM8, for example, could be used to generate up to four different *interrupts* based upon each channel's output-compare register. Even though none of Timer 8's channels are connected to GPIOs, the interrupt routine itself can drive GPIO outputs with software with no loss in precision.

Summary

This chapter applied hardware timers to the task of generating PWM signal outputs suitable for driving RC servo motors. Of course, PWM is not restricted to servos alone. PWM signals may be applied in other ways to take advantage of duty-cycle changes.

The beauty of using hardware timers is that it requires little or no software support once it is configured to run. To change a pulse width or duty cycle requires one small update to the timer, and then the timer goes on its merry way. Hardware timers also offer greater precision since they are not subject to software delays.

EXERCISES

1. In an RC servo signal, what is the period of the signal?

2. Why is the timer input clock frequency 72 MHz on the Blue Pill STM32F103C8T6? Why isn't it 36 MHz?

3. What is changed in the timer to effect a change in the pulse width?

Bibliography

1. STMicroelectronics. Accessed January 12, 2018.
   http://www.st.com/resource/en/reference_manual/
   cd00171190.pdf

CHAPTER 17

PWM Input with Timer 4

The small size of the STM32 makes it a natural application for remote control flying models. Using existing radio controllers, the STM32 could interface with receivers of RC servo signals and perform some special control features from your own imagination.

This chapter includes a demo program that uses Timer 4 to measure an incoming RC servo signal. Because the timer peripheral is doing all the work, the CPU is free to react to the servo readings with more computing power.

The Servo Signal

There is no standard for an RC servo signal, but most seem to use a pulse width of about 0.9 ms at one extreme and about 2.1 ms at the other. The repetition rate is often near 50 Hz, but can be as high as 300 Hz, depending upon manufacturer. Figure 17-1 illustrates the assumed signal that this chapter's demo code will decode.
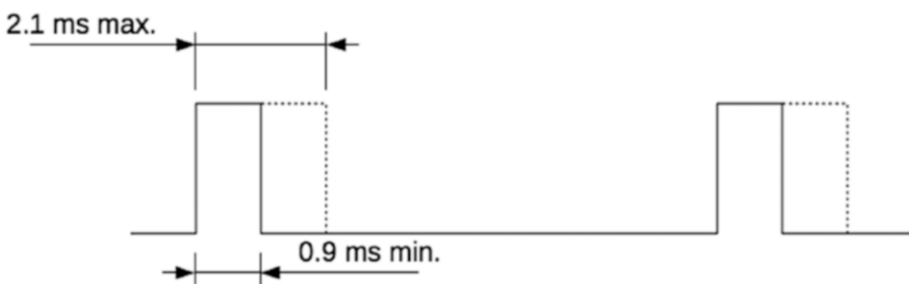
Figure 17-1. *Typical RC servo signal*

Positioning of the servo is governed by the width of the pulse—not the duty cycle. Because of this, some argue that this should not be called PWM (Pulse Width Modulation) at all. The mid-position of the servo is likely to be a pulse 1.5 ms wide.

Signal Voltage

The demo program uses Timer 4, which naturally has its channel 2 on GPIO PB6. This GPIO is 5-volt tolerant. Most servo signal voltages can vary from 4.5 to 6 volts. Use a 2-kohm resistor between the receiver and PB6 to safety limit the current flow. If there is a signal conflict or if the signal rises slightly above 5 volts, the resistor will limit the current to the safe amount of 0.5 mA.

Demo Project

The source code for this project is found here:

```
$ cd ~/stm32f103c8t6/rtos/tim4_pwm_in
```

Now, let's examine the demo software.

GPIO Configuration

The PB6 configuration is pretty routine. Line 45 enables GPIOB's clock, and the remaining lines configure PB6 as an input. Even though this *input* is going into Timer 4, inputs never need to be declared as an alternate GPIO.

```
0045:    // PB6 == TIM4.CH1
0046:    rcc_periph_clock_enable(RCC_GPIOB);  // Need GPIOB clock
0047:    gpio_set_mode(GPIOB,GPIO_MODE_INPUT, // Input
0048:        GPIO_CNF_INPUT_FLOAT,GPIO6);     // PB6=TIM4.CH1
```

Timer 4 Configuration

Like the GPIO, the clock for Timer 4 must be enabled:

```
0043:    rcc_periph_clock_enable(RCC_TIM4);   // Need TIM4 clock
```

Next comes the configuration of the timer itself, shown in Listing 17-1.

Listing 17-1. The Configuration of Timer 4

```
0050:    // TIM4:
0051:    timer_disable_counter(TIM4);
0052:    timer_reset(TIM4);
0053:    nvic_set_priority(NVIC_DMA1_CHANNEL3_IRQ,2);
0054:    nvic_enable_irq(NVIC_TIM4_IRQ);
0055:    timer_set_mode(TIM4,
0056:        TIM_CR1_CKD_CK_INT,
0057:        TIM_CR1_CMS_EDGE,
0058:        TIM_CR1_DIR_UP);
0059:    timer_set_prescaler(TIM4,72);
0060:    timer_ic_set_input(TIM4,TIM_IC1,TIM_IC_IN_TI1);
0061:    timer_ic_set_input(TIM4,TIM_IC2,TIM_IC_IN_TI1);
0062:    timer_ic_set_filter(TIM4,TIM_IC_IN_TI1,TIM_IC_CK_INT_N_2);
0063:    timer_ic_set_prescaler(TIM4,TIM_IC1,TIM_IC_PSC_OFF);
0064:    timer_slave_set_mode(TIM4,TIM_SMCR_SMS_RM);
0065:    timer_slave_set_trigger(TIM4,TIM_SMCR_TS_TI1FP1);
0066:    TIM_CCER(TIM4) &= ~(TIM_CCER_CC2P|TIM_CCER_CC2E
0067:        |TIM_CCER_CC1P|TIM_CCER_CC1E);
0068:    TIM_CCER(TIM4) |= TIM_CCER_CC2P|TIM_CCER_CC2E|TIM_CCER_CC1E;
0069:    timer_ic_enable(TIM4,TIM_IC1);
0070:    timer_ic_enable(TIM4,TIM_IC2);
0071:    timer_enable_irq(TIM4,TIM_DIER_CC1IE|TIM_DIER_CC2IE);
0072:    timer_enable_counter(TIM4);
```

The counter is disabled and reset in lines 51 and 52. Many of the timer's configuration items cannot be changed when it is active. Lines 53 and 54 simply prepare for the Timer 4 interrupts.

The call of line 55 establishes the main elements of TIM4:

- The input to the timer prescaler will be the internal clock.

- The events within the timer will be edge driven.

- The counter will count up.

Line 59 sets the timer's prescaler to 72 so that one clock pulse will occur each microsecond. Recall that the APB1 bus is limited to 36 MHz; therefore, the APB1 prescaler divides by 2 (SYSCLK is 72 MHz). But because the APB1 prescaler is not 1, the timer global prescaler input is the APB1 bus frequency times 2, or 72 MHz.

Lines 60 and 61 indicate that timer inputs IC1 and IC2 are both being directed to timer input 1 (TI1). This fancy bit of configuration means that we can sample the servo signal with a *single* GPIO (PB6) but use *two* differently handled inputs to the timer.

Line 62 establishes a digital input filter that samples the internal clock signal (after timer's private prescaler) divided by two. Line 63 says that the digital filter clock will have no prescaling.

Line 64 specifies that when the PB6 input rises (TI1) the counter should be cleared. The clear happens *after* register TIM4_CCR1 is loaded with the counter's captured value. In this demo, this will be a measure of how long the repeat cycle is.

Line 65 sets the second trigger for Timer 2, causing the timer's current count to be copied to capture register TIM4_CCR2. This happens when the input signal on PB6 falls back to low and thus will measure the time of the pulse width in counter ticks. This signal-change detection is based upon the digitally filtered signal from TI1.

Lines 66 through 68 configure two capture configurations:

- Capture input 1 is *enabled* (TIM_CCER_CC1E), and

- Capture input 1 is active *high* (default), and

- Capture input 2 is *enabled* (TIM_CCER_CC2E), and

- Capture input 2 is active *low* (TIM_CCER_CC2P).

Unfortunately, there are no libopencm3 routines for this at this time, so macro names were used.

Lines 69 and 70 enable the two Timer 4 inputs, and line 71 enables the Timer 4 interrupts for inputs 1 and 2. Finally, line 72 starts the Timer 4 counter.

Task1 Loop

With the timer running, our task enters a loop, which is shown in Listing 17-2. The loop runs leisurely, napping for about a second at line 75. It then toggles the LED on PC13 (as a sign of life).

Listing 17-2. The task1 Demo Loop

```
0019: static volatile uint32_t cc1if = 0, cc2if = 0,
0020:   c1count = 0, c2count = 0;
...
0074:   for (;;) {
0075:       vTaskDelay(1000);
0076:       gpio_toggle(GPIOC,GPIO13);
0077:
0078:       std_printf("cc1if=%u (%u), cc2if=%u (%u)\n",
0079:           (unsigned)cc1if,(unsigned)c1count,
0080:           (unsigned)cc2if,(unsigned)c2count);
0081:   }
```

Lines 78 through 80 report some values of interest:

- CC1IF is the counter value at the end of the cycle, which comes from register TIM4_CCR1. This tells us how long the cycle was in counter ticks. The value displayed in brackets after it is simply the number of times the ISR routine was entered so far.

- CC2IF is the counter value captured when the input signal fell from high to low. This represents the pulse width in counter ticks. The value following in brackets is the ISR count so far.

ISR Routine

The values used by the main loop are updated by the timer's ISR, which is shown in Listing 17-3.

Listing 17-3. The Timer ISR Routine

```
0022: void
0023: tim4_isr(void) {
0024:   uint32_t sr = TIM_SR(TIM4);
0025:
```

```
0026:    if ( sr & TIM_SR_CC1IF ) {
0027:        cc1if = TIM_CCR1(TIM4);
0028:        ++c1count;
0029:        timer_clear_flag(TIM4,TIM_SR_CC1IF);
0030:    }
0031:    if ( sr & TIM_SR_CC2IF ) {
0032:        cc2if = TIM_CCR2(TIM4);
0033:        ++c2count;
0034:        timer_clear_flag(TIM4,TIM_SR_CC2IF);
0035:    }
0036: }
```

The ISR has been enabled for input capture 1 and input capture 2 (lines 69 and 70 of Listing 17-1). When the routine is entered, the timer-status register is read in line 24. If the interrupt is due to the capture 1 event, then flag TIM_SR_CC1IF will be set (line 26). When this is true, the count is captured in line 27 and the interrupt reset in line 29. Line 28 just increments an ISR counter for printing by the main loop.

If the ISR was entered for input capture 2, then the code is similarly executed in lines 32 to 34. The values cc1if, c1count, cc2if, and c2count are the values captured and reported by the main loop (lines 78 to 80 of Listing 17-2). Note that these variables are declared with the volatile attribute because different threads of control are updating/ reading these values.

Demonstration Run

The demonstration consists of hooking up the servo remote control receiver to input GPIO PB6, which is +5-volt tolerant, flashing the code, and running minicom over USB.

First, prepare the software:

```
$ make clobber
$ make
$ make flash
```

Once the software is ready in the MCU flash, it is time to hook up the RC servo receiver. Figure 17-2 illustrates the hookup.
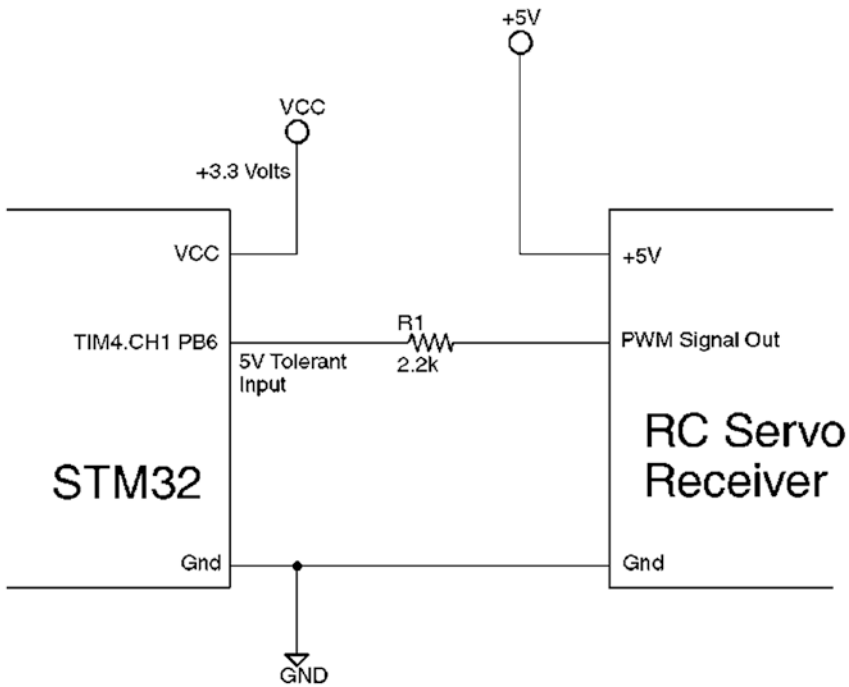
Figure 17-2. *RC servo receiver hookup to STM32*

Resistor R_1 is highly recommended for protection. If for some reason there is a signal conflict, the resistor will limit the current flow to a safe value (3 mA or less). The GPIO input is voltage sensitive, so the resistor won't degrade the signal.

When ready to run, plug the USB cable in and start minicom. I saved my USB settings in a file named "usb" (yours may differ):

```
$ minicom usb
Welcome to minicom 2.7

OPTIONS:
Compiled on Sep 17 2016, 05:53:15.
Port /dev/cu.usbmodemWGDEM1, 19:54:45

Press Meta-Z for help on special keys

cc1if=25174 (176), cc2if=985 (176)
cc1if=25119 (215), cc2if=989 (215)
cc1if=25125 (255), cc2if=974 (255)
cc1if=25172 (294), cc2if=990 (294)
cc1if=25134 (333), cc2if=985 (333)
```

```
cc1if=25183 (372), cc2if=981 (372)
cc1if=25200 (411), cc2if=992 (411)
cc1if=25149 (450), cc2if=990 (450)
cc1if=25339 (489), cc2if=990 (489)
cc1if=24513 (528), cc2if=442 (528)
cc1if=24180 (569), cc2if=209 (569)
cc1if=24135 (610), cc2if=219 (610)
cc1if=24283 (650), cc2if=217 (650)
cc1if=24320 (691), cc2if=208 (691)
cc1if=24265 (732), cc2if=258 (732)
cc1if=25344 (771), cc2if=1027 (771)
cc1if=26232 (809), cc2if=1698 (809)
cc1if=26354 (847), cc2if=1800 (847)
cc1if=26403 (884), cc2if=1871 (884)
cc1if=26495 (921), cc2if=1869 (921)
cc1if=26640 (959), cc2if=1887 (959)
cc1if=26464 (996), cc2if=1896 (996)
cc1if=26489 (1033), cc2if=1868 (1033)
cc1if=26432 (1070), cc2if=1878 (1070)
cc1if=26648 (1107), cc2if=1900 (1107)
cc1if=26431 (1144), cc2if=1883 (1144)
cc1if=26654 (1181), cc2if=1891 (1181)
cc1if=26571 (1219), cc2if=1880 (1218)
cc1if=26566 (1256), cc2if=1889 (1256)
cc1if=26621 (1293), cc2if=1880 (1293)
cc1if=26739 (1330), cc2if=1897 (1330)
```

Session Output

The session output consists of a one-second update of the timer values read. For example, the first line is shown here:

```
cc1if=25174 (176), cc2if=985 (176)
```

The first value shown is the period of the signal. Since the timer is sampling at 1 MHz, this represents a time of:

$$t = \frac{25174}{1000000} = 25.2 \; ms$$

From this figure, we can compute the period of the signal as follows:

$$f = \frac{1}{0.0252} = 39.7 \; Hz$$

The value (176) is the ISR counter value, which is helpful when debugging. This tells us that the ISR was entered 176 times for a timer capture 1 event.

The second value, 985, gives us the pulse width:

$$t = \frac{985}{1000000} = 0.985 \; ms$$

Later on, when the position is changed, we get:

cc1if=26739 (1330), cc2if=1897 (1330)

This represents a pulse width as follows:

$$f = \frac{1897}{1000000} = 1.90 \; ms$$

Timer Inputs

The demonstration illustrated how to accomplish reading the servo receiver, but how did the timer actually accomplish this? Let's clear up the "smoke and mirrors" presentation and examine the inner workings of the input channels.

Take a moment to study Figure 17-3. This presents a somewhat simplified view of the Timer 4 inputs used. Timer 4 has a total of four inputs, but only inputs TI1 and TI2 can be used in this mode of operation.
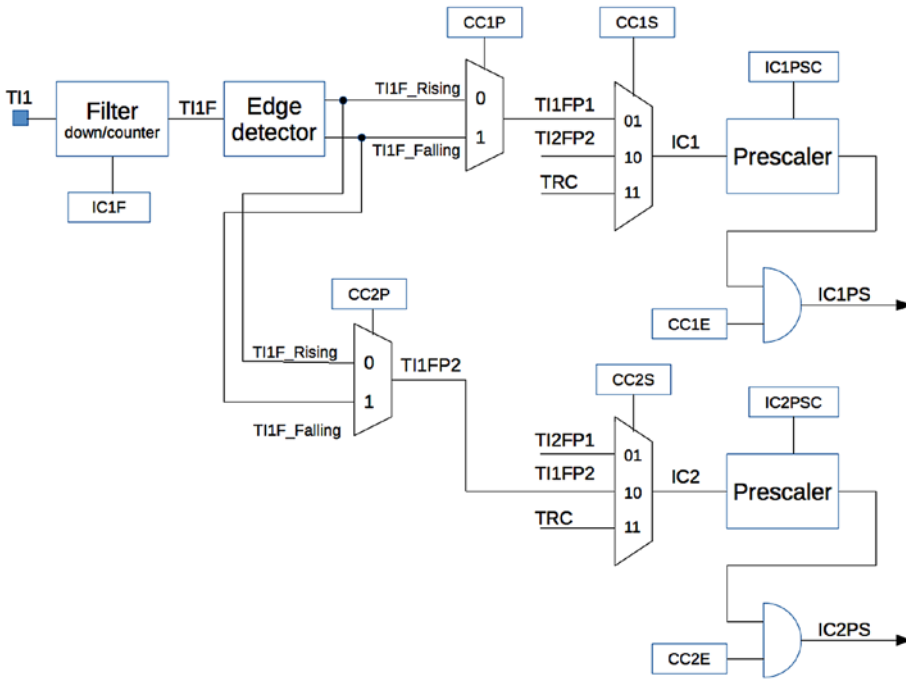
Figure 17-3. *Timer input configuration*

The input signal for the demo enters at TI1 from GPIO PB6. This is timer input channel 1.

TI1 is optionally conditioned by a digital filter controlled by configuration bit IC1F. The value `IC1F` represents a configuration item that is part of a STM32 timer register set. In this case, it is a four-bit value in register `TIM4_CCMR1`. Line 62 of Listing 17-1 sets this value so that the timer's filter counter is 2. Since the clock rate $f_{CK\_INT} = 1\ MHz$, this means that the sampled signal must be stable for two samples before the change appears after the filter at TI1F. This prevents a spurious noise pulse from triggering the timer and spoiling the reading.

Signal TI1F enters an edge detector, producing internal signals TI1F_Rising and TI1F_Falling. Configuration item CC1P chooses which signal polarity to use. Line 60 configures CC1S so that signal TI1FP1 (rising) is used as the input capture signal IC1. When this signal fires (through IC1PS), the timer's counter is copied into the capture 1 register. Line 64 configures the timer such that the timer's counter is *reset after* the capture takes place.

Input signal IC1 can be prescaled, then configured by IC1PSC. Line 63 disabled this optional feature so that there is no prescaling. Lines 68 and 69 enable CC1E, allowing the signal IC1PS to activate the capture 1 event. This particular capture measures the period of the PWM signal in the demo.

But we're not done yet.

The configuration uses input channel 2 (IC2), derived from the signal sampled by *channel 1*. Configuration item CC2S in this mode causes the TI1F_Rising or TI1F_Falling (channel 1) signal to be used instead of the normal TI2 input. This is useful when measuring PWM input because we need to capture different events from the same signal. The remainder of the I2CPS chain is otherwise the same as for I1CPS, except that it drives the capture 2 event. Because IC2 is the opposite polarity (falling edge) arranged by CC2S, I2CPS can cause the capture of the counter when the signal falls. This gives us the counter at the point at which the pulse width returns to low.

Summary

This chapter demonstrated how the STM32 timer can be used to effortlessly measure the pulse width and period of a signal. In the demo, only 39 x 2 interrupts were executed to capture the period and pulse width every second. The ISR code is quite minimal, leaving valuable CPU cycles available to perform other useful work.

There are many other timer-input features that remain unexplored. The reader is encouraged to scour the STM32 reference manual RM0008 for more ideas.

EXERCISES

1. Why does the timer have a digital filter available on its inputs?

2. When does the timer reset in PWM input mode?

3. Where does the IC2 input signal come from in PWM input mode?

CAN Bus

The development of the CAN bus (controller area network) began in 1983 at Robert Bosch GmbH as a way to standardize communications between components. Prior to this, automotive ECUs (engine control units) each had their own proprietary systems, which required a lot of point-to-point wiring. In 1986, Bosch released the developed CAN protocol at the SAE Congress (Society of Automotive Engineers).[1] In 1991 the CAN 2.0 protocol was published by Bosch and was adopted in 1993 as the international standard (ISO 11898). Since then, automobiles have used the protocol for communication and to reduce wiring harness sizes by use of the bus.

Having CAN bus capability in the STM32 device makes it attractive for automotive or control applications. Even though this chapter's demonstration will model the control system of a car, it will become apparent that the CAN bus need not be restricted to automotive applications. Model aircraft, drones, and model railway systems are just some of the potential hobby applications.

The CAN Bus

Imagine that you have the task of reducing the bulk of the wiring harness for a new car model to be manufactured. This vehicle has several electronic control units at the front, center, and rear of the vehicle, and each requires communication with the others, including the master control unit, which is perhaps located behind the dashboard. How do you reduce the number of wires needed?

Almost since the beginning, manufacturers have reduced the wiring by using the metal body of the car as the negative terminal for the battery-return current. However, the *control* of the load has been traditionally handled by switching the +12-volt power on and off to the brake or signal lamp, for example. This requires a separate wire in the harness for each load to be controlled.

Additionally, with electronic control units, you now must also supply lines of communication. With units that communicate with most or all of the other units, the number of wired connections explodes.

The solution to the harness problem is to adopt a bus system. Every control unit that communicates is attached to the same bus and sends messages over the bus. With this configuration, you need the following:

- a power line (+12 volts)

- one or a pair of bus signal lines

- a negative return path for power (metal car body)

If we assume a pair of bus signal lines then we only need three wires to power and communicate with all devices connected to it. Figure 18-1 illustrates a hypothetical bus system and power connections.
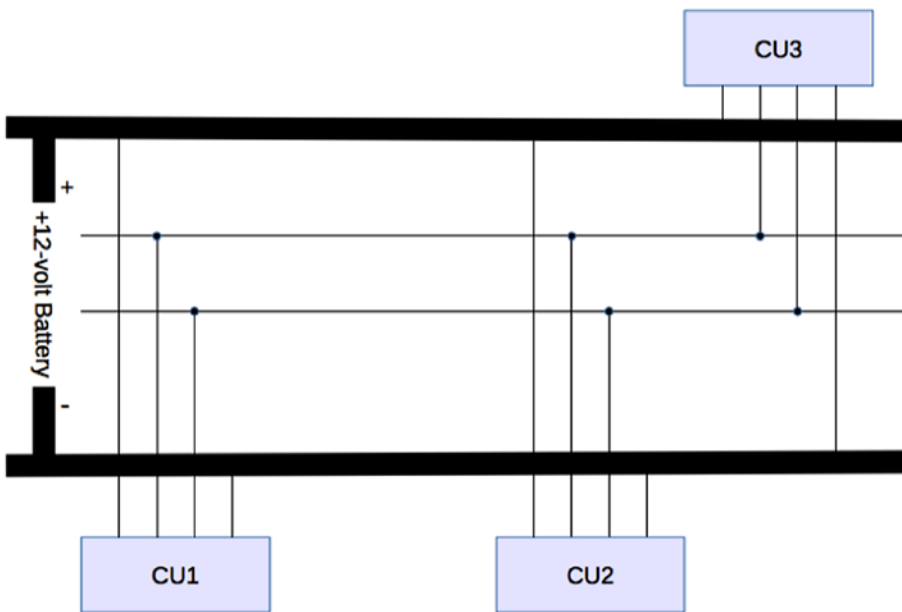
Figure 18-1. *Hypothetical automotive bus system*

Using this arrangement, any control unit CUx can communicate with any other control unit. If control unit CU3 is located at the rear of the vehicle, then it could also control light bulbs located at the rear based upon messages passed on the bus. CU3 can switch nearby lamps by switching the +12-volt power with short runs of wire to the

unit itself. While automobiles may not do this presently for brake and signal lights, it is technically possible to do so. Sometimes there may be overriding safety concerns—it is important that brake lights don't fail as a result of software errors, for example. The bus design does, however, allow widespread communication and provides control with lighter wiring.

Differential Signals

High-speed linear CAN bus is defined by the ISO 11898-2 signal format, with an example signal shown in the upper part of Figure 18-2. The signal pair consists of a CAN H (high) and a CAN L (low) line, the latter of which idles near the 2.5-volt level. The idle state is known as the *recessive* logic level.
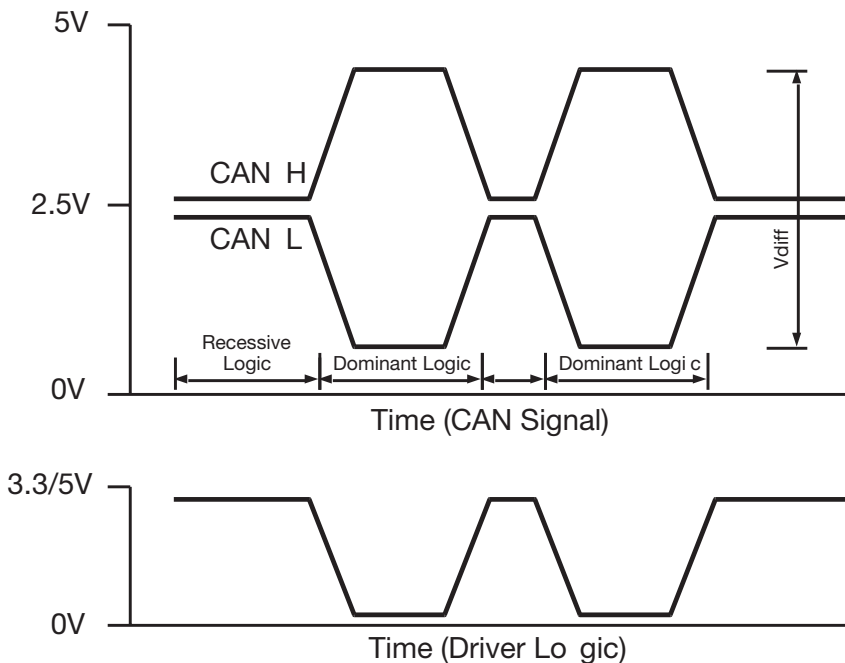
Figure 18-2. *High-speed linear CAN bus and driver-signal formats*

The active state of the signal is known in the standard as the *dominant* logic level. The dominant logic state has CAN H near +5 volts and CAN L near 0 volts. The differential signal is required for high-speed communication with noise immunity. The state of the logic is determined by how much CAN H differs from CAN L (V_{diff} in the figure). To prevent signal reflections, the high-speed linear bus is terminated at each end with 120 Ω of resistance.

The lower half of Figure 18-2 shows the single-ended *driver* signal. For our purposes, this is the signal that leaves the STM32 and enters the driver chip. From a driver-signal perspective, the dominant logic level is *low*, while the recessive level is *high*. High can be 5 volts or 3.3 volts, depending upon the logic levels being utilized.

The driver circuit requires only one wire, but this limits it to short runs because of signal reflections and noise. The differential bus design, on the other hand, allows for longer runs of several meters.

Dominant/Recessive

Recessive and dominant signal forms have already been described for the CAN bus. But what is the significance of these terms, *recessive* and *dominant*?

A recessive signal is a relaxed form of the signal. In differential signalling, the CAN H and CAN L signals reach their idle state through a resistive network. This is the natural state for the signal at rest. For the single-ended driver signal, it represents a high level that is achieved by a pull-up resistor. This too is the rest state of the driver signal.

A dominant signal, however, is driven from its rest state to its active state through the work of transistors. For the differential bus signals, the CAN H line is pulled high by a high-side transistor. Simultaneously, the CAN L line is pulled low by a low-side transistor in the on state. The driver signal likewise goes from a pulled-high state to a pulled-low state by a low-side transistor in conduction. In other words, the driver signal is driven low by an open-drain transistor in the active state.

The differential bus is like the single-ended driver signal except that there are mirror copies of each signal. They both idle near the middle when at rest (recessive) but are pulled away from each other when made active (dominant).

Now imagine two units driving a common bus. It is easiest to think in terms of the single-ended driver signal, but do realize that the principle also applies to the differential bus. Table 18-1 illustrates a truth table for two drivers connected to the bus.

Table 18-1. *Truth Table for Bus Arbitration*

Driver 1	Driver 2	Bus Result	Description
recessive	recessive	recessive	Bus is idle
dominant	recessive	dominant	Dominant state
recessive	dominant	dominant	Dominant state
dominant	dominant	dominant	Dominant state

Essentially, the bus state becomes the logical OR of all drivers connected to the bus. When any driver applies the dominant bit state, the bus takes the dominant state. Only when no driver is driving the bus does the bus remain in the recessive state. It should be obvious now why the states are named recessive and dominant.

Bus Arbitration

In bus systems such as I2C, there is one master and multiple slaves. The slave device is not allowed to speak until requested by the master. In multi-master I2C there has to be an arbitration procedure that works out which master is allowed to proceed in the event that two or more devices collide trying to transmit at the same time. This tends to be a complicated problem that can lead to bus lockups.

The CAN bus, on the other hand, permits every connected device to speak. Thus, collision avoidance also requires arbitration. The way it is done for the CAN bus is unique and relies on the principle of recessive and dominant states. Figure 18-3 illustrates how the recessive and dominant bits interact.
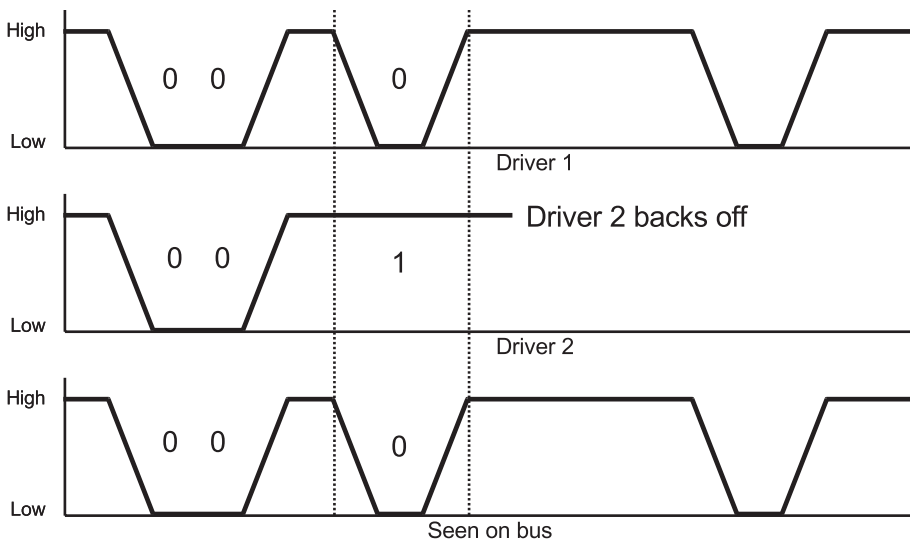
Figure 18-3. *CAN bus arbitration*

Arbitration proceeds as follows:

1. Driver 1 and Driver 2 start a communication at the same time by sending a message ID.

2. The first two bits are dominant (zero), so all devices connected to the bus see the two zeros.

3. Driver 1 sends the third bit as dominant (zero), while Driver 2 attempts to send a recessive (one) bit. The bus sees only a dominant bit (zero).

4. Driver 2, realizing that its recessive bit was "stomped on," backs off since it has lost the arbitration process. Driver 1's message was not harmed and can proceed.

Arbitration illustrates the purpose of the dominant bits. Each device continues to transmit while listening to the bus. If any device sends a recessive bit but reads it back as a dominant bit, it means that arbitration for the bus was lost. The losing device(s) then cancel their transmission, allowing the winner to proceed with the message unharmed.

Synchronization

The arbitration procedure just presented is a simplified explanation for a more complicated reality. How do several transmitters communicate in lockstep?

Before the message ID is transmitted, there is an SOF bit (start of frame) sent. It is a dominant (zero) bit, so no listening device will miss the start-of-message indication that it is. Thus, when the SOF bit is received, the sending device on the same bus can cancel an attempt to send a message if it is not ready. If the device is ready, it synchronizes its clock and attempts to send the message ID.

After the SOF bit, each message begins with a message ID. Depending on the version of the protocol, the message ID is 11 or 29 bits in length. The arbitration procedure determines who will win based upon the message ID being sent. Recall that dominant bits win over recessive bits. Consequently, a message ID value of all zero bits is the highest priority message.

Message Format

The basic CAN message format is provided in Table 18-2. There is also an extended format that differs slightly, which the reader is encouraged to research.

Table 18-2. *CAN Bus Message Format (Non-extended)*

Field Name	Bit Length	Description
SOF	1	Start of Frame
ID	11	Message ID/Priority
RTR	1	Remote Transmission Request: 0 for data frames, 1 for remote request
IDE	1	Identifier extension: 0 for 11-bit format, 1 for 29-bit
Reserved	1	Must be 0
DLC	4	Data Length Code: 0 to 8 bytes
Data	0–64	Transmitted data (DLC sets length)
CRC	15	Cyclic redundancy check
CRC delimiter	1	Must be 1 (recessive)
ACK slot	1	Transmitter sends 1 (recessive), receiver(s) respond with 0 (dominant)
ACK delimiter	1	Must be 1 (recessive)
EOF	1	End of Frame: 1 (recessive)

Frame field RTR (Remote Transmission Request) has a provision for requesting a transmission from the remote device. When the RTR bit is recessive, this indicates a response from a device. In these messages, the DLC (Data Length Code) is not used, and no data is sent.

The field IDE (Identifier extension bit) identifies whether an extended ID message format is used. In this chapter, the demo uses the 11-bit format, using the dominant bit setting to indicate this.

The DLC field indicates the data length (in non-RTR response) in *bytes*. The DLC field is then followed by the indicated number of data bytes.

Near the frame end, the CRC (Cyclic Redundancy Check) field exists so that garbled messages can be disregarded.

Finally, at the frame's end is an ACK bit field. This bit is transmitted as a recessive bit (1) so that if any device receives the message with CRC intact, the receiving device will clamp the bus using a dominant state during that bit time. This allows the transmitter to see that at least one device received the message ok. Generally speaking, if one device receives the message ok, then all devices did. Note that all receiving devices are permitted to respond with the ACK simultaneously.

If, on the other hand, no devices received the message, the ACK bit will remain at the recessive state as transmitted. This indicates to the transmitter that the transmission failed. This part of the protocol can make CAN bus driver development a little more difficult because you need at least one other device on the bus to ACK the sent message. The only other way to test the sending of a CAN message is to use an internal loop-back feature of the STM32 peripheral.

STM32 Limitation

The STM32 CAN bus peripheral uses SRAM to store messages. Unfortunately, the design of the STM32F103 MCU is such that CAN and USB share the same area of memory. For this reason, CAN and USB cannot be used at the same time.

It is otherwise possible to disable one device and use the other and vice versa, but this doesn't appear to be practical. For this reason, the demonstration will use the UART for communication.

Demonstration

There is considerably more that could be described about CAN bus protocol and its extensions. The interested reader is encouraged to seek out other books on the subject. The focus of this chapter, however, is to illustrate how to use the STM32's CAN bus peripheral in ways that are simple enough to get started.

In this demo, we are going to implement three hypothetical EU (electronic units). I'll refer to them as EU1 through EU3 and avoid the acronym ECU, which is normally known as an engine control unit. The units are:

1. EU1, dashboard control unit (has UART interface). This provides lamp controls and reads temperature from the rear EU3.

2. EU2, a rear controller unit responsible for parking, signal, and brake lamps

3. EU3, a front controller unit responsible for front parking and signal lamps

A full demonstration thus requires three STM32 MCUs. However, if you have two, you can at least partially demonstrate the operation, though three works best. Leave out the front EU2 if necessary. But with the low price of the Blue Pill, why limit yourself to only two units?

Software Build

The software directory for the demonstration is located at the following:

```
$ cd stm32f103c8t6/rtos/can
```
Take a moment now to recompile it:

```
$ make clobber
$ make
```

This will compile three executables:

```
$ ls *.elf
front.elf    main.elf    rear.elf
```

UART Interface

EU1 is the main control unit that will simulate what will reside behind the dashboard of our hypothetical vehicle (module `main.c`). The demonstration is configured for the following serial-port parameters, which must agree with minicom or the terminal program of your choice:

- Baud rate: 115,200

- Data bits: 8

- Parity: None

- Stop bit: 1

- Hardware flow control: RTS/CTS

See Table 10-1 for the connection details.

Students, please note that hardware flow control requires RTS and CTS wires to be connected to your TTL serial adapter and STM32. Once connected, your terminal program must also be configured to use hardware flow control. If any detail is incorrect, no communication will occur. If the flow control is not operational for some reason, then you may see lost data and garbage.

If the hardware flow control presents issues or your TTL serial adapter lacks the required RTS/CTS signals, change the following source line in `main.c` from `open_uart(1,115200,"8N1","rw",1,1);` to the following:

`open_uart(1,9600,"8N1","rw",0,0);`

Later, after recompiling, if there seems to be some data loss, reduce the baud rate even further. Otherwise, the lower baud rate of 9,600 should be alright.

MCU Flashing

There are three MCUs to flash for this demonstration:

1. EU1: `main.c`

2. EU2: `front.c`

3. EU3: `rear.c`

Let's flash the devices (after the build). You'll obviously need to hook the programmer up to each device in turn. If you need to reflash these, you will likely need to change the boot0 jumper temporarily.

```
$ make flash
$ make flash_front
$ make flash_rear
```

Demo Bus

To make things easier, this demonstration uses a short but single-wire CAN bus (SWCAN) using a 4.7-kohm pull-up resistor. In this configuration, each STM32 MCU has its CAN_RX and CAN_TX lines tied together and connected to the common bus line. Normally, these connections would go to a CAN bus driver chip like PCA82C251, with separate RX and TX connections. Search for "PCA82C251 datasheet PDF" for more details about the chip.

When wiring the demo, make special note of the fact that the main.c MCU uses *different* CAN connections from the others. This was done so that the 5-volt-tolerant UART connections could be used, permitting 5-volt USB-TTL serial adapters to be used. See Figure 18-4.
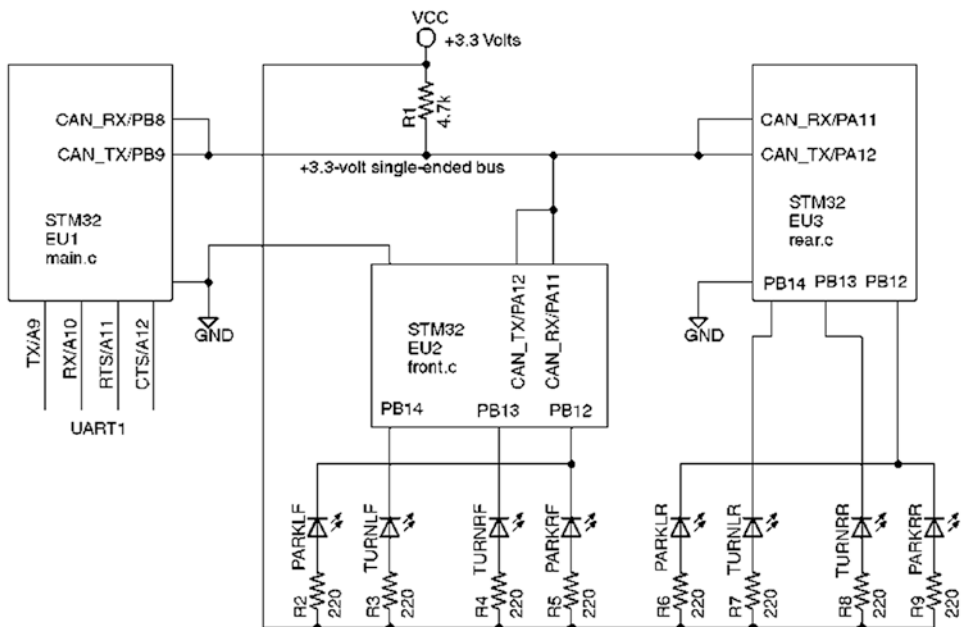
Figure 18-4. Demonstration CAN bus hookup

Resistor R_1 is a required pull-up resistance necessary to establish the recessive state. Since the MCU is a 3.3-volt device, our recessive state will be near +3.3 volts. Because of the pull-up resistor, we configure the GPIO for CAN_TX with `GPIO_CNF_OUTPUT_ALTFN_OPENDRAIN` (emphasis on *open drain*).

The other MCUs share the bus and the grounds. Don't forget to tie those grounds together. Any message sent by one MCU is received by all of the others.

The LEDs are representative of automotive lamps for signal, brake, and parking lights. The rear signal lamps operate as both brake and signal lamps.

Session Run

The demo can be set up on a breadboard. Figure 18-5 shows the author's own arrangement, with the power supplied by a MB102 PCB. This provides +5 volts to each of the Blue Pill +5-volt *inputs*, resulting in each Blue Pill's on-board regulator supplying +3.3 volts to the remainder of the system.
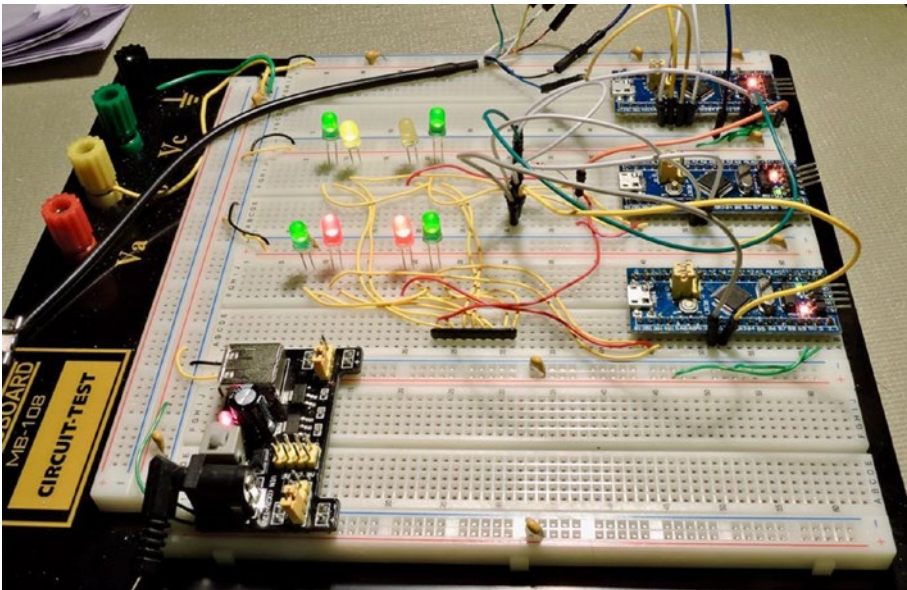
Figure 18-5. *Breadboard setup of CAN demo*

In the center of the photo, you can see the SIP9 resistor array I used for the 220-ohm LED resistors. These single inline package (nine pins) arrays conveniently replace up to nine individual resistors. Pin 1 of the SIP9 resistor is common and goes to the +3.3-volt

supply line. To turn on an LED, the connected GPIOs sink the current to ground. If not using the SIP9 part, simply use individual resistors as shown (R_2 to R_9).

The top of the photo shows the USB-UART device cable and connections to the top MCU (running firmware `main.c`). This represents the in-dash controller EU1. The bottom MCU is the rear EU3 unit running firmware `rear.c`.

The LEDs are arranged with the front automotive lamps toward the top and the rear lamps toward the bottom of the photo. The outer green LEDs represent the parking lights. The inner yellow LEDs represent the front turn signals, while the rear red LEDs represent the red turn signals and brake lights.

The single-ended CAN bus is located just to the right of the LEDs, with mostly white DuPont wires connecting the CAN_RX and CAN_TX from each MCU. Within that mess of wires is one 4.7-kohm resistor pulled up to +3.3 volts.

Once everything is ready, connect your USB-TTL serial device to your PC and start minicom (or equivalent). Once that is ready, power up your breadboard. The main MCU should respond to your serial link as follows:

```
Welcome to minicom 2.7

OPTIONS:
Compiled on Sep 17 2016, 05:53:15.
Port /dev/cu.usbserial-A703CYQ5, 20:38:01

Press Meta-Z for help on special keys

Car simulation begun.

Menu:

  L - Turn on left signals
  R - Turn on right signals
  P - Turn on parking lights
  B - Activate brake lights
  Lower case the above to turn OFF

  V - Verbose mode (show received messages)

CAN Console Ready:
> _
```

The serial link shows a menu allowing you to control various functions of your hypothetical vehicle. If things are working correctly the front and rear MCU on-board LEDs (PC13) should flash about once per second. If you see it blink twice and stop, this indicates a bus connection problem. I experienced this problem when I forgot that CAN GPIOs are different from the main and the front and rear units. Recheck your wiring.

Press a capital "P" to turn on the parking lights. If all went well, your parking lights are now lit, and the console also returned a temperature:

```
CAN Console Ready:
> P
> Temperature: +24.73 C
>
```

The temperature was sent to the main unit from the rear unit. To turn the parking lights off, press a lowercase "p." The parking lights should go dark immediately.

If you now press capital "B," the brake lights should come on (red in my setup). Pressing lowercase "b" turns them off again.

Press capital "L" or "R" to turn on the left or right turn signals, respectively. Notice that the front and rear signals blink in unison even though controlled by two separate MCUs. Press the lowercase "l" or "r" to turn the signal off again. You can also turn on four-way flashers by enabling left *and* right.

Last of all, enable a turn signal—say, left—by pressing capital "L." Then, press capital "B" to enable the brake lights. Now the left turn signal blinks, but the right rear remains lit to indicate a brake light. Turning the turn signal lamp off should leave the two rear brake lights lit.

CAN Messages

The messages are mainly sent from the main EU1 to the front and rear units. After each lamp request, the main unit also sends a message to the rear requesting a temperature (it sets the RTR bit to request a reply). When the rear unit receives a message with the RTR flag true, it takes the temperature and transmits it to the bus. All others can read this message, but only the main unit uses the information.

The other messages are sent to enable/disable a given lamp. To allow the signal lamps to flash in unison, there is also a flash message sent.

Synchronicity

When the signal lamps are flashing, they look very synchronized to the human eye. But just how synchronized are they? Using an oscilloscope, the turning on and off of a signal lamp varied by about ±850 μsec and changes over time. All MCUs receive the message into their peripherals at the same time, but FreeRTOS does preemptive scheduling. Since 1-ms ticks are being used, timing could be off by up to 1 ms.

What if you need greater accuracy for a factory control application? One approach would be to increase the timer tick frequency (reducing the time slice). Other improvements are possible in the application software. For example, the ISR routine could notify a waiting task. There is no single answer to this problem. It often comes down to how important the issue is and how much effort you are willing to expend to obtain it.

Summary

This chapter has focused on introducing some CAN bus concepts and a demo circuit. Running the demo proved that it is possible to have near real-time control over other MCUs using short CAN messages. Additionally, it proves the concept of a shared bus where there are no master and slave devices. Finally, it is seen that the STM32 is capable of applying CAN communications in both single-wire or differential bus modes (differential with the help of a driver chip).

Because of the size and complexity of this project, the software for this demo will be described in the next chapter.

Bibliography

1. CSS Electronics. 2018. "CAN Bus Explained—A Simple Intro."
   Accessed January 13, 2018. https://www.csselectronics.com/
   screen/page/simple-intro-to-can-bus/language/en.

CHAPTER 19

CAN Bus Software

The CAN bus demonstration in the previous chapter illustrated three STM32 MCUs sharing messages on a common bus. None were masters and none were slaves. All of this was orchestrated with the help of the STM32 CAN bus peripheral and the libopencm3 device driver.

This chapter will discuss the use of the libopencm3 driver API so that you can build CAN bus applications of your own. When combined with the use of FreeRTOS, you will have a convenient environment from which to program more-complex creations.

Initialization

The project source modules are located in the following directory:

```
$ cd ~/stm32f103c8t6/rtos/can
```

The most demanding part of setting up the CAN bus peripheral is the configuration and initialization of it. Listing 19-1 illustrates the initialize_can() function that was provided in source module canmsgs.c.

Listing 19-1. The CAN Initialization Code

```
0090: void
0091: initialize_can(bool nart,bool locked,bool altcfg) {
0092:
0093:   rcc_periph_clock_enable(RCC_AFIO);
0094:   rcc_peripheral_enable_clock(&RCC_APB1ENR, RCC_APB1ENR_CAN1EN);
0095:
0096:   /**************************************************
0097:    * When:
0098:    *    altcfg     CAN_RX=PB8,  CAN_TX=PB9
0099:    *    !altcfg    CAN_RX=PA11, CAN_TX=PA12
```

```
0100:     **************************************************/
0101:    if ( altcfg ) {
0102:        rcc_periph_clock_enable(RCC_GPIOB);
0103:        gpio_set_mode(GPIOB,GPIO_MODE_OUTPUT_50_MHZ,
                 GPIO_CNF_OUTPUT_ALTFN_OPENDRAIN,
                 GPIO_CAN_PB_TX);
0104:        gpio_set_mode(GPIOB,GPIO_MODE_INPUT,GPIO_CNF_INPUT_FLOAT,
                 GPIO_CAN_PB_RX);
0105:
0106:        gpio_primary_remap(    // Map CAN1 to use PB8/PB9
0107:                AFIO_MAPR_SWJ_CFG_JTAG_OFF_SW_OFF, // Optional
0108:                AFIO_MAPR_CAN1_REMAP_PORTB);
0109:    } else  {
0110:        rcc_periph_clock_enable(RCC_GPIOA);
0111:        gpio_set_mode(GPIOA,GPIO_MODE_OUTPUT_50_MHZ,
                 GPIO_CNF_OUTPUT_ALTFN_OPENDRAIN,GPIO_CAN_TX);
0112:        gpio_set_mode(GPIOA,GPIO_MODE_INPUT,
                 GPIO_CNF_INPUT_FLOAT,GPIO_CAN_RX);
0113:
0114:        gpio_primary_remap( // Map CAN1 to use PA11/PA12
0115:            AFIO_MAPR_SWJ_CFG_JTAG_OFF_SW_OFF, // Optional
0116:        AFIO_MAPR_CAN1_REMAP_PORTA);
0117:    }
0118:
0119:    can_reset(CAN1);
0120:    can_init(
0121:        CAN1,
0122:        false,    // ttcm=off
0123:        false,    // auto bus off management
0124:        true,     // Automatic wakeup mode.
0125:        nart,     // No automatic retransmission.
0126:        locked,   // Receive FIFO locked mode
0127:        false,    // Transmit FIFO priority (msg id)
0128:        PARM_SJW, // Resynch time quanta jump width (0..3)
0129:        PARM_TS1, // segment 1 time quanta width
0130:        PARM_TS2, // Time segment 2 time quanta width
```

```
0131:        PARM_BRP, // Baud rate prescaler for 33.333 kbs
0132:        false,     // Loopback
0133:        false);   // Silent
0134:
0135:    can_filter_id_mask_16bit_init(
0137:        0,                        // Filter bank 0
0138:        0x000 << 5, 0x001 << 5,   // LSB == 0
0139:        0x000 << 5, 0x001 << 5,   // Not used
0140:        0,                        // FIFO 0
0141:        true);
0142:
0143:    can_filter_id_mask_16bit_init(
0145:        1,                        // Filter bank 1
0146:        0x010 << 5, 0x001 << 5,   // LSB == 1 (no match)
0147:        0x001 << 5, 0x001 << 5,   // Match when odd
0148:        1,                        // FIFO 1
0149:        true);
0150:
0151:    canrxq = xQueueCreate(33,sizeof(struct s_canmsg));
0152:
0153:    nvic_enable_irq(NVIC_USB_LP_CAN_RX0_IRQ);
0154:    nvic_enable_irq(NVIC_CAN_RX1_IRQ);
0155:    can_enable_irq(CAN1,CAN_IER_FMPIE0|CAN_IER_FMPIE1);
0156:
0157:    xTaskCreate(can_rx_task,"canrx",400,NULL,
0158:            configMAX_PRIORITIES-1,NULL);
0158: }
```

This function provides initialization in the following basic steps:

1. The AFIO subsystem's clock is enabled (line 93). This is necessary so that we can chose which GPIOs are used for the CAN bus ports.

2. The CAN bus peripheral's clock is enabled (line 94). This is required for the peripheral to function.

335

3. The appropriate GPIO has its clock enabled (line 102 or 110, depending upon the configuration chosen by Boolean argument `altcfg`).

4. The GPIO output mode is chosen for CAN_TX (line 103 or 111).

5. The GPIO input mode is chosen for CAN_RX (line 104 or 112).

6. The AFIO mapping is chosen for the CAN_TX and CAN_RX lines (lines 106 to 108, or lines 114 to 116).

7. The libopencm3 routine `can_reset()` is called to initialize and configure the CAN bus peripheral (lines 119 to 133).

8. CAN filter bank 0 is configured in lines 135 to 141 to determine where certain messages should go.

9. CAN filter bank 1 is configured in lines 143 to 149 to determine where other messages should go.

10. A FreeRTOS receive queue named `canrxq` is created in line 151.

11. The STM32 NVIC has two interrupt channels enabled in lines 153 and 154.

12. The CAN bus peripheral has the FIFO message pending interrupts enabled for FIFO 0 and 1 (line 155).

13. Finally, a receiving task is created in line 157.

There is obviously quite a bit of detail in this procedure. Let's break down some of the steps.

can_init()

The `can_init()` function is provided by libopencm3 and requires several arguments to configure the device. Let's examine the calling arguments in more detail. The arguments provided are as follows:

1. Argument CAN1 indicates which peripheral to use. There is only one available for the STM32F103C8T6.

2. This argument is supplied with `false` to indicate that we are not using time-triggered communication mode.

3. This argument is supplied with `false` to indicate that we are not using automatic bus-off mode (if too many errors occur, the bus can be auto-disabled).

4. This argument is supplied with `true` to indicate that we want automatic wakeup mode, should the MCU be put to sleep.

5. This argument is supplied with our called argument `nart`. When true, this indicates that we do not want the CAN peripheral auto-retransmit when an error is detected.

6. The argument is supplied with our called argument `locked`. When true, it means that when a receive FIFO becomes full, no new message will displace an existing message (the FIFO is locked). When false, new messages can displace existing messages when the FIFO is full.

7. This argument is supplied as `false` to have outgoing messages be given priority according to their message ID. Otherwise, messages are transmitted in chronological order.

8. `PARM_SJW`

9. `PARM_TS1`

10. `PARM_TS2` defines CAN synchronization parameters.

11. `PARM_BRP` is declared as `78` and `canmsgs.h` so that the effective baud rate is 33.333 kbs.

12. The second-to-last argument is supplied with `false` to disable the loopback capability of the peripheral.

13. The last argument is supplied with `false` so that it operates in "normal mode." When in silent mode, the peripheral can receive remote data but cannot initiate messages (it is silent).

CAN Receive Filters

The CAN bus peripheral has the ability to filter messages of interest. If you imagine a large set of bus-connected communicators, it becomes evident that a lot of message traffic will be received. Normally, not every node is interested in all messages. Processing every message would eat away at the available CPU budget.

The CAN peripheral supports two FIFO queues for receiving messages. With the help of filtering, this demonstration arranges for even-numbered message IDs to land in FIFO 0 and odd-numbered messages in FIFO 1. This is arranged by the configuration of filter banks 0 and 1.

The call to can_filter_id_mask16bit_init() in lines 135 to 141 arranges a set of messages to land in FIFO 0 (line 140). Argument two in this example is declaring the configuration of filter bank 0 (line 137). The last argument (true) simply enables the filter.

Arguments three (line 138) and four (line 139) define the actual filter ID value and bit mask to use. These are 16-bit filters, but the filter is 32 bits wide. For this reason, two identical filters are used:

- 0x000 is the resulting ID to match against after applying the mask.

- 0x001 is the bit mask to be applied to the ID before comparing.

Both of these arguments must be shifted up five bits to the left in order to left justify the 11-bit identifiers in the 16-bit field.

In the second configured filter (lines 143 to 149) we have the same mask value (0x001) but compare two different ID values:

- 0x010 is a "no match" ID.

- 0x001 is the odd value after masking.

If the mask 0x001 is applied to an ID, it matches 0x001 when the ID is odd. However, no matter what ID is supplied after being masked with 0x001, it will never match the value 0x010 given. This is simply another way of disabling the second unused filter.

As configured, a message will always be odd or even and will wind up in one of the FIFO receive queues (CAN peripheral FIFO).

There are several other possibilities for specifying filters, including using 32-bit values so that extended ID values can be compared. The reader is encouraged to review the libopencm3 API documentation and the STMicroelectronics RM0008 reference manual, section 24.7.4, for more information.

CAN Receive Interrupts

Each CAN FIFO (first in first out queue) has its own ISR. This permits the designer to allocate different interrupt priorities to each FIFO queue. In this demo, we treat both identically, so Listing 19-2 illustrates the interlude routines used to redirect the code to one common function named can_rx_isr().

Listing 19-2. The CAN Receive Interlude ISRs

```
0058: void
0059: usb_lp_can_rx0_isr(void) {
0060:     can_rx_isr(0,CAN_RF0R(CAN1)&3);
0061: }

0067: void
0068: can_rx1_isr(void) {
0069:     can_rx_isr(1,CAN_RF1R(CAN1)&3);
0070: }
```

The macros CAN_RF0R() and CAN_RF1R() allow the code to determine the length of the FIFO queues. The common code for the CAN receive ISR is shown in Listing 19-3.

Listing 19-3. The Common CAN Receive ISR

```
0029: static void
0030: can_rx_isr(uint8_t fifo,unsigned msgcount) {
0031:     struct s_canmsg cmsg;
0032:     bool xmsgidf, rtrf;
0033:
0034:     while ( msgcount-- > 0 ) {
0035:         can_receive(
0036:             CAN1,
0037:             fifo,                 // FIFO # 1
0038:             true,                 // Release
0039:             &cmsg.msgid,
0040:             &xmsgidf,             // true if msgid is extended
0041:             &rtrf,                // true if requested transmission
0042:             (uint8_t *)&cmsg.fmi, // Matched filter index
```

```
0043:                &cmsg.length,           // Returned length
0044:                cmsg.data,
0045:                NULL);                  // Unused timestamp
0046:            cmsg.xmsgidf = xmsgidf;
0047:            cmsg.rtrf = rtrf;
0048:            cmsg.fifo = fifo;
0049:            // If the queue is full, the message is lost
0050:            xQueueSendToBackFromISR(canrxq,&cmsg,NULL);
0051:        }
0052: }
```

The general flow of the code is as follows:

1. Receive the message (lines 35 to 45).

2. Queue the message to FreeRTOS queue canrxq (line 50).

3. Repeat until there are no more messages (line 34).

To understand the other details, we need to know about the structures involved. These are illustrated in Listing 19-4.

Listing 19-4. Message Structures Found in canmsgs.h

```
0020: struct s_canmsg {
0021:    uint32_t    msgid;        // Message ID
0022:    uint32_t    fmi;          // Filter index
0023:    uint8_t     length;       // Data length
0024:    uint8_t     data[8];      // Received data
0025:    uint8_t     xmsgidf : 1;  // Extended message flag
0026:    uint8_t     rtrf : 1;     // RTR flag
0027:    uint8_t     fifo : 1;     // RX Fifo 0 or 1
0028: };
0029:
0030: enum MsgID {
0031:    ID_LeftEn = 100,          // Left signals on/off (s_lamp_en)
0032:    ID_RightEn,               // Right signals on/off (s_lamp_en)
0033:    ID_ParkEn,                // Parking lights on/off (s_lamp_en)
0034:    ID_BrakeEn,               // Brake lights on/off (s_lamp_en)
```

```
0035:    ID_Flash,                 // Inverts signal bulb flash
0036:    ID_Temp,                  // Temperature
0037:    ID_HeartBeat = 200,       // Heartbeat signal (s_lamp_status)
0038:    ID_HeartBeat2             // Rear unit heartbeat
0039: };
0040:
0041: struct s_lamp_en {
0042:    uint8_t      enable : 1;  // 1==on, 0==off
0043:    uint8_t      reserved : 1;
0044: };
0045:
0046: struct s_temp100 {
0047:    int          celciusx100;  // Degrees Celcius x 100
0048: };
0049:
0050: struct s_lamp_status {
0051:    uint8_t      left : 1;     // Left signal on
0052:    uint8_t      right : 1;    // Right signal on
0053:    uint8_t      park : 1;     // Parking lights on
0054:    uint8_t      brake : 1;    // Brake lines on
0055:    uint8_t      flash : 1;    // True for signal flash
0056:    uint8_t      reserved : 4;
0057: };
```

Essentially, the message is received into the struct s_canmsg. See lines 35 to 45
of Listing 19-3. Some parts have to be loaded and then copied to the structure. For
example, the structure member xmsgidf is a 1-bit-sized member, so it is received in
a local variable named xmsgidf (line 40) and then copied to cmsg.xmsgidf in line 46.
Other members are copied into the structure in lines 47 and 48. By the time execution
continues at line 50 the structure is fully populated and then copied to the queue.
Notice that the FreeRTOS routine called is xQueueSendToBackFromISR(); i.e., ending in
"FromISR()." This is necessary since special arrangements often need to be made in an
ISR due to their asynchronous nature.

The main payload is carried in the data[8] array, and its active length is given by
member length in this program. Our application uses truly small messages.

The message is indicated by the message ID. This is documented in the following:

```
0030: enum MsgID {
0031:    ID_LeftEn = 100,      // Left signals on/off (s_lamp_en)
0032:    ID_RightEn,           // Right signals on/off (s_lamp_en)
0033:    ID_ParkEn,            // Parking lights on/off (s_lamp_en)
0034:    ID_BrakeEn,           // Brake lights on/off (s_lamp_en)
0035:    ID_Flash,             // Inverts signal bulb flash
0036:    ID_Temp,              // Temperature
0037:    ID_HeartBeat = 200,   // Heartbeat signal (s_lamp_status)
0038:    ID_HeartBeat2         // Rear unit heartbeat
0039: };
```

Pop quiz: Which is the highest-priority message in this set?

Answer: ID_LeftEn (with value 100).

Why? Because this is the lowest (defined) message ID in the set. Recall that with the nature of CAN dominant bits the lowest message ID will always win an arbitration contest.

These are message types used by our demo program. Message ID value ID_HeartBeat is an "I'm alive" message from the front controller, while ID_HeartBeat2 is a similar message from the rear controller. Our demo doesn't do anything with these messages when received, but with more code the main controller could warn if the front or rear controller wasn't sending a message at regular intervals.

Message ID values ID_LeftEn, ID_RightEn, ID_ParkEn, and ID_BrakeEn indicate a lamp-control message. The struct s_lamp_en carries the intended action. Its member enable indicates whether the message is to turn on or off a lamp. This data is carried in the data[] array member of s_canmsg:

```
0041: struct s_lamp_en {
0042:    uint8_t     enable : 1;     // 1==on, 0==off
0043:    uint8_t     reserved : 1;
0044: };
```

The message ID_Temp is used both to request a temperature and to receive one. The main control unit will request a temperature reading by sending ID_Temp, with the member rtrf set to true. When the rear control unit receives this message, it will take a reading and reply with the rtrf flag set to false. The temperature returned is carried in the data[] member as a struct s_temp100:

```
0046: struct s_temp100 {
0047:   int        celciusx100;    // Degrees Celcius x 100
0048: };
```

Application Receiving

Once the ISR queues the data message s_canmsg, something must pull messages out of the queue. In module canmsgs.c there is a task defined for this purpose:

```
0076: static void
0077: can_rx_task(void *arg __attribute((unused))) {
0078:     struct s_canmsg cmsg;
0079:
0080:     for (;;) {
0081:         if ( xQueueReceive(canrxq,&cmsg,portMAX_DELAY) == pdPASS )
0082:             can_recv(&cmsg);
0083:     }
0084: }
```

This small task simply pulls messages from canrxq that were queued by the ISR. If there are no messages in the queue, the task will block forever due to the timeout argument portMAX_DELAY in line 81. If a message is successfully pulled from the queue, the application function can_recv() is called with it (not to be confused with the libopencm3 routine named can_receive()).

Processing the Message

The application is made aware of incoming CAN messages when the can_recv() function is called by the module canmsgs.c. This is performed outside of an ISR call, so most programming functions should be safe to use. Listing 19-5 illustrates the function declared in rear.c.

Listing 19-5. Processing a Received CAN Message in the Application (from rear.c)

```
0119: void
0120: can_recv(struct s_canmsg *msg) {
0121:     union u_msg {
0122:         struct s_lamp_en    lamp;
0123:     } *msgp = (union u_msg *)msg->data;
0124:     struct s_temp100 temp_msg;
0125:
0126:     gpio_toggle(GPIO_PORT_LED,GPIO_LED);
0127:
0128:     if ( !msg->rtrf ) {
0129:         // Received commands:
0130:         switch ( msg->msgid ) {
0131:         case ID_LeftEn:
0132:         case ID_RightEn:
0133:         case ID_ParkEn:
0134:         case ID_BrakeEn:
0135:         case ID_Flash:
0136:             lamp_enable((enum MsgID)msg->msgid,msgp->lamp.enable);
0137:             break;
0138:         default:
0139:             break;
0140:         }
0141:     } else {
0142:         // Requests:
0143:         switch ( msg->msgid ) {
0144:         case ID_Temp:
0145:             temp_msg.celciusx100 = degrees_C100();
0146:             can_xmit(ID_Temp,false,false,sizeof temp_msg,&temp_msg);
0147:             break;
0148:         default:
0149:             break;
0150:         }
0151:     }
0152: }
```

The message is passed into can_recv() with a pointer to the s_canmsg structure, which is populated with the received data. Because the data member msg->data is interpreted based upon its message ID, the union u_msg is declared in lines 121 to 123. This permits the programmer to access the msg->data array as a struct s_lamp_en when it is needed.

To show sign of life, line 126 toggles the onboard LED (PC13). Normal messages will not have the msg->rtrf flag set (line 128). In this case, we expect the usual lamp commands (lines 130 to 137).

Otherwise, when msg->rtrf is true, this represents a request for the rear module to read the temperature and respond with it (lines 143 to 150). The function degrees_C100() returns the temperature in degrees Celsius times one hundred. This is simply transmitted by line 146. Note that the third argument of the call is the RTR flag, which is sent as false (this is the response). The function can_xmit() is declared in canmsgs.c, not to be confused with the libopencm3 routine can_transmit().

Be mindful that can_recv() is called as part of another task. This requires safe inter-task communication.

Sending CAN Messages

Sending messages is easy, since we simply need to call the libopencm3 routine can_transmit():

```
0018: void
0019: can_xmit(uint32_t id,bool ext,bool rtr,uint8_t length,void *data) {
0020:
0021:   while ( can_transmit(CAN1,id,ext,rtr,length,(uint8_t*)data) == -1 )
0022:       taskYIELD();
0023: }
```

In the Blue Pill hardware there is only one CAN bus controller, so CAN1 is hardcoded as argument one here. The message ID is passed through id, the extended address flag is passed through ext, and the request flag rtr is passed as argument four. Lastly, the length of the data and the pointer to the data are supplied. If the call fails by returning -1, taskYIELD() is called to share the CPU time. The call will fail if the sending CAN bus peripheral queue is full.

This brings up an important point to keep in mind. Upon successful return from `can_transmit()`, the caller cannot assume that the message has been sent *yet*. Recall that in our configuration we declared whether messages are sent in priority sequence or in chronological sequence. But the bus can also be busy, delaying the actual sending of our messages. Further, if our message(s) are lower priority (higher message ID values) than others on the bus, our CAN peripheral must wait until it can win bus arbitration.

Summary

The remainder of the demo is routine C code. The reader is encouraged to review it. By packaging the CAN bus application API in modules `canmsgs.c` and `canmsgs.h`, the task of writing the application becomes easier. It also saves time by using common tested code.

This demo has only scratched the surface of what can be done on the CAN bus. Some folks may want to listen in on their vehicles, but a word of caution is warranted. Some CAN bus designs, like GMLAN (General Motors Local Area Network), can include 12-volt signal spikes for use as a wakeup signal. There are likely a number of other variations of that theme.

The CAN bus has been applied to a number of other applications, such as factory and elevator controls. After working with this demo project, you can entertain new design ideas.

EXERCISES

1. How many FIFOs are supported by the STM32F103 CAN peripheral?

2. How many filter banks are supported by the CAN peripheral?

3. When a pair of filters must be supplied, but only one is needed, what are two ways to accomplish this?

4. What is the RTR flag and what is its purpose?

CHAPTER 20

New Projects

Starting a new project from scratch can be a lot of work. That is why this chapter is focused on helping you get started with the minimum of drudgery. I'm also going to point you to a few details that have been ignored for the sake of simplicity that might be important to your project. This will leave you in the driver's seat.

Project Creation

The first step in a new project is to create its subdirectory, `Makefile`, then import the FreeRTOS source modules and a starting configuration file named `FreeRTOSConfig.h`. Yes, you can do this manually or with a script, but the provided `Makefile` will do this for you.

First, locate the right starting directory:

```
$ cd ~/stm32f103c8t6/rtos
```

Think of a good subdirectory name for your project that doesn't already exist. For this example, we'll call it `myproj`. To create a project named myproj, perform the following `make` command:

```
$ make -f Project.mk PROJECT=myproj
...bunch of copies etc...
**************************************************************
Your project in subdirectory myproj is now ready.

1. Edit FreeRTOSConfig.h per project requirements.
2. Edit Makefile SRCFILES as required. This also
   chooses which heap_*.c to use.
3. Edit stm32f103c8t6.ld if necessary.
4. make
5. make flash
```

6. make clean or make clobber as required
****

If you now produce a recursive list of your subdirectory, you will see that it has been populated with several files:

```
$ ls -R ./myproj
FreeRTOSConfig.h   Makefile      main.c          rtos
stm32f103c8t6.ld

./myproj/rtos:
FreeRTOS.h          heap_1.c      list.h          portmacro.h
task.h              LICENSE       heap_2.c        mpu_prototypes.h
projdefs.h          tasks.c       StackMacros.h   heap_3.c
mpu_wrappers.h      queue.c       timers.h        croutine.h
heap_4.c            opencm3.c     queue.h         deprecated_definitions.h
heap_5.c            port.c        semphr.h        event_groups.h
list.c              portable.h    stdint.readme
```

Makefile

Listing 20-1 illustrates the Makefile that will be created for you. This file should normally be edited slightly to reflect the source modules that you will use. This Makefile uses several macros to define the overall project. Let's examine those now.

Listing 20-1. Default Project Makefile

```
0001: ##########################################################
0002: #  Project Makefile
0003: ##########################################################
0004:
0005: BINARY     = main
0006: SRCFILES    = main.c rtos/heap_4.c rtos/list.c rtos/port.c \
                    rtos/queue.c rtos/tasks.c rtos/opencm3.c
0007: LDSCRIPT   = stm32f103c8t6.ld
0008:
0009: # DEPS      = # Any additional dependencies for your build
0010: # CLOBBER   += # Any additional files to be removed with \
```

```
                         "make clobber"
0011:
0012: include ../../Makefile.incl
0013: include ../Makefile.rtos
0014:
0015: ##########################################################
0016: #  NOTES:
0017: #    1. remove any modules you don't need from SRCFILES
0018: #    2. "make clean" will remove *.o etc., but leaves *.elf, *.bin
0019: #    3. "make clobber" will "clean" and remove *.elf, *.bin etc.
0020: #    4. "make flash" will perform:
0021: #        st-flash write main.bin 0x8000000
0022: ##########################################################
```

Macro BINARY

This macro defines the name of your compiled executable. By default, this is set to main so that main.elf is produced when the project is built. By all means, change this to something more exciting.

Macro SRCFILES

This macro defines the name of the source files that will be compiled into the final executable main.elf. The default is to include the following source files:

- main.c (main should match the name used in the BINARY macro)
- rtos/heap_4.c
- rtos/list.c
- rtos/port.c
- rtos/queue.c
- rtos/tasks.c
- rtos/opencm3.c

The module `main.c` (or otherwise named) is the module you will write and develop. The remaining modules are *support* modules, which will be discussed later on. Some of these are *optional*. For example, if you don't use FreeRTOS message queues, you can leave out the module `rtos/queue.c` (with the appropriate changes to `FreeRTOSConfig.h`).

If you have additional source files (in addition to `main.c`), add them to the `SRCFILES` list. They too will be compiled and linked into the final build.

Macro LDSCRIPT

The provided `Makefile` sets this to `stm32f103c8t6.ld`. This points to a file in your project directory, which you have already seen in Chapter 9, "Overlays." Many projects can use this file unchanged. If your project has special needs like overlays, it can be altered.

Macro DEPS

If you have special dependencies, you can define them with this macro. For example, if you have a text file like `mysettings.xml`, which affects the build of `main.elf`, then to force a rebuild of `main.elf` add the following:

```
DEPS = mysettings.xml
```

Macro CLOBBER

The `make` files have been written to support some basic targets, including `clobber`. For example:

```
$ make clobber
```
This command eliminates files that were built and are unnecessary to keep. For example, all object files (`*.o`) and executables (`*.elf`) would be deleted as a cleanup operation. This also guarantees that everything is built from *scratch* the next time you perform a `make`.

Sometimes a build procedure creates other objects that can be removed after the build is complete. If a `file.dat` is generated by the build process and you want it cleaned up after a clobber, add it to the macro:

```
CLOBBER   = file.dat
```

Included Makefiles

To reduce the footprint of the project `Makefile` and centralize other definitions, two more `Makefiles` are included in your project file:

- `../../Makefile.incl (~/stm32f103c8t6/Makefile.incl)`

- `../Makefile.rtos (~/stm32f103c8t6/rtos/Makefile.rtos)`

The first of these defines macros and rules for building projects. If you need to make project-wide enhancements to the `make` rules, this is the place to start.

The second of these simply adds the subdirectory `./rtos` to be searched for `include` files for FreeRTOS builds:

```
TGT_CFLAGS      += -I./rtos -I.
TGT_CXXFLAGS    += -I./rtos -I.
```

Header Dependencies

The `DEPS` macro described earlier adds dependencies for building `main.elf`. What if you have another header file named `myproj.h` that, if changed, would cause a recompile of `main.o`? This can be done with the usual `Makefile` dependency rule added:

```
main.o: myproj.h
```

This informs the `make` command that `main.c` should be recompiled into `main.o` if the datestamp of file `myproj.h` is newer. That might save you from chasing bugs related to a header-file change when main.o was not rebuilt when it should have been.

Compile Options

Sometimes a special compile option is needed for certain modules. In the OLED project, this was used to suppress some compiler warnings from a third-party source module:

```
ugui.o:  CFLAGS += -Wno-parentheses
```

This compile option is only added to the compile of `ugui.o` to suppress warnings about brackets that should be added for clarity.

Flashing 128k

If you haven't already done so, change to your project subdirectory. After you build your project with

```
$ cd ./myproj
$ make
```

you need to program the STM32 flash storage. By default, the make command is set up to do this with the following:

```
$ make flash
arm-none-eabi-objcopy -Obinary main.elf main.bin
/usr/local/bin/st-flash  write main.bin 0x8000000
...
```

The first step in this is to convert the elf file (main.elf) to a binary image (main.bin). The ARM version of the objcopy utility performs that duty. After that, the st-flash utility is used to program the STM32.

Most, if not all, STM32F103C8T6 chips will support flashing to 128k. You will need to have the newer version st-flash utility installed. To flash more than 64k, perform the following:

```
$ make bigflash
arm-none-eabi-objcopy -Obinary main.elf main.bin
/usr/local/bin/st-flash --flash=128k write main.bin 0x8000000
...
```

The new option --flash=128k is supplied to the st-flash utility to disregard the device ID and flash up to 128k worth of memory.

The real question is whether all STM32F103C8T6 chips do indeed support 128k. All four of my units did, purchased from different eBay sources. In fact, there is only one online reported instance of this not working. Was this pilot error? Or is it that only the lower 64k is factory tested and guaranteed? If someone knows the answer, I would like to know.

FreeRTOS

An important part of your project is the FreeRTOS components. Some are optional, while others are mandatory. Let's look at each in turn.

rtos/opencm3.c

This module was written by the author and is not actually part of FreeRTOS. It is required to connect the libopencm3 framework into FreeRTOS. The module is shown in Listing 20-2.

Listing 20-2. Source Module ~/ stm32f103c8t6/rtos/opencm3.c

```
0001: /* Warren W. Gay VE3WWG
0002:  *
0003:  * To use libopencm3 with FreeRTOS on Cortex-M3 platform, we must
0004:  * define three interlude routines.
0005:  */
0006: #include "FreeRTOS.h"
0007: #include "task.h"
0008: #include <libopencm3/stm32/rcc.h>
0009: #include <libopencm3/stm32/gpio.h>
0010: #include <libopencm3/cm3/nvic.h>
0011:
0012: extern void vPortSVCHandler( void ) __attribute__ (( naked ));
0013: extern void xPortPendSVHandler( void ) __attribute__ (( naked ));
0014: extern void xPortSysTickHandler( void );
0015:
0016: void sv_call_handler(void) {
0017:    vPortSVCHandler();
0018: }
0019:
0020: void pend_sv_handler(void) {
0021:    xPortPendSVHandler();
0022: }
0023:
0024: void sys_tick_handler(void) {
```

```
0025:    xPortSysTickHandler();
0026: }
0027:
0028: /* end opncm3.c */
```

As the source code indicates, these libopencm3 functions call into FreeRTOS. For example, function sys_tick_handler() is invoked by libopencm3 when the system timer tick interrupt occurs. But the call to xPortSysTickHandler() is a FreeRTOS function that handles the system tick operations.

rtos/heap_4.c

This is the FreeRTOS module used throughout this book. However, there are actually multiple choices possible. Quoted from the www.freertos.org web page[1]:

- heap_1 – the very simplest; does not permit memory to be freed

- heap_2 – permits memory to be freed, but not does coalescence adjacent free blocks

- heap_3 – simply wraps the standard malloc() and free() for thread safety

- heap_4 – coalescences adjacent free blocks to avoid fragmentation; includes absolute address placement option

- heap_5 – as per heap_4, with the ability to span the heap across multiple non-adjacent memory areas

Some of these source modules can be affected by the FreeRTOSConfig.h macro setting configAPPLICATION_ALLOCATED_HEAP. Simply swap the rtos/heap_4.c mentioned in the Makefile with the module of your choice.

Required Modules

In addition to the dynamic memory module rtos/heap_*.c, the following are normally required modules for FreeRTOS:

- rtos/list.c (internal list support)

- rtos/port.c (portability support)

- rtos/queue.c (queue and semaphore support)

- rtos/tasks.c (task support)

Depending upon the options chosen in your FreeRTOSConfig.h file, in your project directory, you may be able to exclude one or two modules for a smaller build. For example, if you don't use queue, mutex, or semaphore support, you can avoid linking in rtos/queue.c.

FreeRTOSConfig.h

This is your project-level FreeRTOS configuration file. This include file contains macro settings that affect the way that the FreeRTOS modules are compiled into your project. They may also affect any macro calls invoked by your application. You should make clobber before you rebuild if any of the values in that file are changed. Remember that you are building both the O/S and the application together.

Listing 20-3 provides a partial listing of what is contained in the FreeRTOSConfig.h file. The first section configures items such as the CPU clock rate (configCPU_CLOCK_HZ). Others determine features like preemption (configUSE_PREEMPTION).

One pair of important macro settings are configCPU_CLOCK_HZ and configSYSTICK_CLOCK_HZ. For use with libopencm3, using a 72 MHz clock, you normally want to configure configSYSTICK_CLOCK_HZ as follows:

```
#define configSYSTICK_CLOCK_HZ ( configCPU_CLOCK_HZ / 8 )
```

If you get this wrong, a program using vTaskDelay() or other time-related functions will be incorrect. You can check this by running the demo in stm32f103c8t6/rtos/blinky2. When incorrectly configured, the blink will not be half a second.

Listing 20-3. Partial Listing of FreeRTOSConfig.h, Used in the RTC Project

```
/*-----------------------------------------------------------
 * Application-specific definitions.
 *
 * These definitions should be adjusted for your particular hardware and
 * application requirements.
 *
 * THESE PARAMETERS ARE DESCRIBED WITHIN THE 'CONFIGURATION' SECTION OF THE
 * FreeRTOS API DOCUMENTATION AVAILABLE ON THE FreeRTOS.org WEB SITE.
```

```
 *
 * See http://www.freertos.org/a00110.html.
 *----------------------------------------------------------*/

#define configUSE_PREEMPTION              1
#define configUSE_IDLE_HOOK               0
#define configUSE_TICK_HOOK               0
#define configCPU_CLOCK_HZ                    ( ( unsigned long ) 72000000 )
#define configSYSTICK_CLOCK_HZ   ( configCPU_CLOCK_HZ / 8 )
#define configTICK_RATE_HZ       ( ( TickType_t ) 1000 )
#define configMAX_PRIORITIES     ( 5 )
#define configMINIMAL_STACK_SIZE ( ( unsigned short ) 128 )
#define configTOTAL_HEAP_SIZE    ( ( size_t ) ( 17 * 1024 ) )
#define configMAX_TASK_NAME_LEN  ( 16 )
#define configUSE_TRACE_FACILITY       0
#define configUSE_16_BIT_TICKS         0
#define configIDLE_SHOULD_YIELD        0
#define configUSE_MUTEXES              1
#define configUSE_TASK_NOTIFICATIONS  1
#define configUSE_TIME_SLICING         1
#define configUSE_RECURSIVE_MUTEXES    0

/* Co-routine definitions. */
#define configUSE_CO_ROUTINES          0
#define configMAX_CO_ROUTINE_PRIORITIES ( 2 )

/* Set the following definitions to 1 to include the API function, or zero
to exclude the API function. */

#define INCLUDE_vTaskPrioritySet        1
#define INCLUDE_uxTaskPriorityGet       1
#define INCLUDE_vTaskDelete             0
#define INCLUDE_vTaskCleanUpResources   0
#define INCLUDE_vTaskSuspend            1
#define INCLUDE_vTaskDelayUntil         1
#define INCLUDE_vTaskDelay              1
```

The macro `configTOTAL_HEAP_SIZE` is important to configure if you encounter the following error:

```
section '.bss' will not fit in region 'ram'
```
   As defined:

```
#define configTOTAL_HEAP_SIZE       ( ( size_t ) ( 17 * 1024 ) )
```

FreeRTOS will allocate 17k to the heap. But as you develop your killer application and use more static memory areas, the remaining SRAM storage may shrink to the point where the heap won't fit. This will prevent the link step from completing. What you can do is reduce the heap size until it builds. Advanced users can look at the memory map produced to see how much you can re-increase the heap. Or you could just guess by increasing the heap until it fails to build.

Other configuration macros like `INCLUDE_vTaskDelete` simply determine whether that level of support should be compiled into FreeRTOS for your application. If you never delete a task, why include code for it?

All of these configuration options are documented at the FreeRTOS website and in their fine free manual.

User Libraries

It is common practice to place commonly used routines like USB or UART drivers in a library. Once you develop these, you want to reuse them. You may have noticed that this was done in some of our demo projects in this book. By default, all programs include headers from `~/stm32f103c8t6/rtos/libwwg/include` and link to the library directory `~/stm32f103c8t6/rtos/libwwg`, linking with `libwwg.a`.

Within the directory `~/ stm32f103c8t6/rtos/libwwg/src` are some source modules that go into that static library. These get compiled and the object modules placed into `libwwg.a`. But there is a problem here that you should be aware of.

These are all compiled against the following header file:

```
~/stm32f103c8t6/rtos/libwwg/src/rtos/FreeRTOSConfig.h
```
   This is likely different from your project-level file `FreeRTOSConfig.h`. If the configurations differ in a material way, the best approach is to copy the needed source files into your project directory and add them to your `Makefile` (`SRCFILES`). When you do that, you guarantee that the subroutines use the `FreeRTOSConfig.h` that the rest of your application is compiled with.

Once again, this is related to the fact that every build of your application also includes the build of the operating system. Much of it is driven by macros, so header files play a significant role.

Rookie Mistakes

One rookie mistake that *all of us* get bitten by from time to time is to make a change to a structure in a header file that affects modules that don't get recompiled. When a structure is altered, the offsets of members change. A previously compiled module will continue to use the old member offsets.

Ideally, the `Makefile` would list *every* dependency or have it generated and then included. However, this isn't always done, or done perfectly enough, especially during frantic project development.

If you have changed a struct (or class in C++), it is recommended practice to perform a `make clobber` first so that everything is recompiled from scratch. In huge projects, this approach is impractical. But for small projects, this ensures that all modules are compiled from the same headers.

Do you have a bug that doesn't make sense? The impossible is happening? Perhaps you need to rebuild your project from scratch to make sure that you aren't chasing toolchain problems.

Summary

This chapter has prepared you for creating your own STM32 projects using FreeRTOS. You've reviewed the FreeRTOS modules that go into your build, as well as the glue module that links libopencm3 to FreeRTOS. With the ability to configure `FreeRTOSConfig.h`, you can direct how your project is built.

Bibliography

1. "Memory Management [More Advanced]." FreeRTOS—Memory management options for the FreeRTOS small footprint, professional grade, real time kernel (scheduler). Accessed February 1, 2018. https://www.freertos.org/a00111.html.

EXERCISES

1. What does the `make` macro BINARY define?

2. What is the purpose of the header file `FreeRTOSConfig.h`?

3. How do you add compiler option `-O3` only to the compile of module `speedy.c`?

4. What is the main disadvantage of using `heap_1`?

Troubleshooting

No matter how trivial the project or how sure we are about developing a project, we inevitably run into the need for troubleshooting. The need is so often greater for an embedded computing project because you don't have the luxury of a core file dump to be analyzed like you would under Linux. You might also not have a display device at the point of the error.

In this chapter, we'll first look at the debugging facilities that are available to the STM32 platform. Then, some troubleshooting techniques will be examined, along with other resources.

Gnu GDB

The Gnu GDB debugger is quite powerful and worth taking the time to learn. Using the ST-LINK V2 USB programmer, it is possible to access the STM32 from your desktop and step through the code, examining memory and registers and setting breakpoints. The first step is to get the GDB server up and running.

GDB Server

Open another terminal session where you can run your GDB server. The `st-util` command will have been installed with your `st-flash` software install. If you launch `st-util` without the programmer being plugged into the USB port, your session will appear something like this:

```
$ st-util
st-util 1.3.1-4-g9d08810
2018-02-08T21:09:22 WARN src/usb.c: Couldn't find any ST-Link/V2 devices
```

If you see this, check that your ST-LINK V2 programmer is connected and plugged in.

If your programmer is plugged in but it doesn't see the STM32 device attached to it, your session will appear something like this:

```
$ st-util
st-util 1.3.1-4-g9d08810
2018-02-08T21:10:52 INFO src/usb.c: -- exit_dfu_mode
2018-02-08T21:10:52 INFO src/common.c: Loading device parameters....
2018-02-08T21:10:52 WARN src/common.c: unknown chip id! 0xe0042000
```

If your device *is* attached, unplug the programmer immediately to avoid damage and recheck the wiring. If you are using individual DuPont wires between the programmer and the STM32, it is easy to make a mistake. To avoid that, I recommend that you make a custom cable for this purpose.

If everything goes well, you should have a session like the following:

```
$ st-util
st-util 1.3.1-4-g9d08810
2018-02-08T21:07:18 INFO src/usb.c: -- exit_dfu_mode
2018-02-08T21:07:18 INFO src/common.c: Loading device parameters....
2018-02-08T21:07:18 INFO src/common.c: Device connected is: F1 \
    Medium-density device, id 0x20036410
2018-02-08T21:07:18 INFO src/common.c: SRAM size: 0x5000 bytes (20 KiB), \
    Flash: 0x20000 bytes (128 KiB) in pages of 1024 bytes
2018-02-08T21:07:18 INFO src/gdbserver/gdb-server.c: Chip ID is 00000410,\
    Core ID is  1ba01477.
2018-02-08T21:07:18 INFO src/gdbserver/gdb-server.c: Listening at *:4242..
```

From this, we observe that we are connected to an F1 device (STM32F103) and that it found 20K bytes of static RAM. Depending upon your device and the version of your `st-util` command installed, it may show only 64K bytes of flash, or, as it does here, it may show 128K bytes instead. Last of all, notice that it is listening at `*:4242`. The asterisk indicates that it is listening on all interfaces at port 4242.

When you want to terminate this server, press ^C (Control-C).

Remote GDB

With the `st-util` server running, it is now possible to use GDB to connect to that server to start a debug session. Let's use the RTC project as a working example:

```
$ cd ~/stm32f103c8t6/rtos/rtc
```

It is critical that you use the version of GDB that matches your compiler tools. Most of you will likely be using `arm-none-eabi-gdb`, though it may differ by your install. Since this is tedious to type, you may want to use a shell alias for this purpose:

```
$ alias g='arm-none-eabi-gdb'
```

This allows you to just type "g" to invoke it. I'll list the command in full in this chapter, but do use the alias to save typing if you like. Just start up the command to get started:

```
$ arm-none-eabi-gdb
GNU gdb (GNU Tools for ARM Embedded Processors 6-2017-q2-update)
7.12.1.20170417-git
Copyright (C) 2017 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
...
For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb)
```

At this point, we have not yet attached to the `st-util` server. The next step is optional but is necessary if you want to have source-level symbols and debugging in your session:

```
(gdb) file main.elf
Reading symbols from main.elf...done.
```

Notice that it confirms that the symbols were extracted from the file `main.elf` in the current directory. Next, connect to the `st-util` server:

```
(gdb) target extended-remote :4242
Remote debugging using :4242
0x08003060 in ?? ()
(gdb)
```

As a shortcut, no IP address is typed before the `:4242`. This implies that we are connecting through the local loopback `127.0.0.1:4242`. The connection is confirmed with the message "Remote debugging using :4242."

Now, let's load the program into flash:

```
(gdb) load main.elf
Loading section .text, size 0x30e8 lma 0x8000000
Loading section .data, size 0x858 lma 0x80030e8
Start address 0x8002550, load size 14656
Transfer rate: 8 KB/sec, 7328 bytes/write.
```

The `st-util` server will automatically flash the file's image into flash (note that it loads it starting at address `0x8000000`). There are also data areas programmed into flash starting at address `0x80030e8`. The startup code will locate this and copy that data to its proper location in SRAM before the `main()` function is called.

Next, we set a breakpoint for the `main()` function:

```
(gdb) b main
Breakpoint 1 at 0x800046c: file main.c, line 252.
(gdb)
```

If we don't set a breakpoint, the software will run away and execute when we launch it. Setting the breakpoint at `main` allows all the initialization to run, but it stops at the first statement in the `main()` program. Note that the breakpoint command will fail if you leave out the `file` command (earlier) because it won't know about the symbol `main`.

Now, let's start the program:

```
(gdb) c
Continuing.

Breakpoint 1, main () at main.c:252
252     main(void) {
(gdb)
```

The program has started and then paused at the breakpoint that we set. Now, we can step over source statements with the "n" (next) GDB command:

```
(gdb) n
254     rcc_clock_setup_in_hse_8mhz_out_72mhz(); // Use this for "blue
pill"
```

```
(gdb) n
256     rcc_periph_clock_enable(RCC_GPIOC);
(gdb) n
257     gpio_set_mode(GPIOC,GPIO_MODE_OUTPUT_50_MHZ,
                    GPIO_CNF_OUTPUT_PUSHPULL,GPIO13);
(gdb)
```

This has allowed us to step over these three statements. If you want to trace inside any function, issue the "s" (step) GDB command instead.

To just run the program from this point forward, use the "c" (continue) GDB command:

```
(gdb) c
Continuing.
```

Notice that no new (GDB) prompt is returned. To interrupt the program and regain control, press ^C (Control-C):

```
^C
Program received signal SIGTRAP, Trace/breakpoint trap.
0x08000842 in xPortPendSVHandler () at rtos/port.c:403
403         __asm volatile
(gdb)
```

Where the program is interrupted at will vary. To view the call stack, use the bt (backtrace) GDB command:

```
(gdb) bt
#0  0x08000842 in xPortPendSVHandler () at rtos/port.c:403
#1  <signal handler called>
#2  0x08000798 in prvPortStartFirstTask () at rtos/port.c:270
#3  0x080008d6 in xPortStartScheduler () at rtos/port.c:350
Backtrace stopped: Cannot access memory at address 0x20005004
(gdb)
```

This tells us that we interrupted execution inside of the FreeRTOS scheduler code.

To exit GDB, type the command "quit."

GDB Text User Interface

To make debugging sessions more convenient, GDB supports a few different layouts. Figure 21-1 is the layout obtained by typing the command "`layout split.`" This gives you both the source code and the assembler-level instruction view.

Figure 21-1. *GDB "layout split" display*

Other views are possible. For example, to trace what happens in assembler language programs you'll want to use the "layout regs" view, shown in Figure 21-2. This view shows the assembler language instructions as well as the register content. Changed registers are highlighted. Unfortunately, the small terminal window size used for Figure 21-2 doesn't do it justice. When you use a wider terminal window, you will see all of the registers.

Figure 21-2. *GDB "layout regs" view (full register set displayed with wider terminal window)*

There is quite a bit more to GDB than can be described here. An investment in reading the GDB manual or online tutorials can save you time in the long run.

Peripheral GPIO Trouble

You write a new program using the UART peripheral, which requires the use of a GPIO output. You configure it, yet the output doesn't work. Hopefully, this book has already prepared you for the answer. What is wrong with this code fragment?

```
rcc_periph_clock_enable(RCC_GPIOA);
rcc_periph_clock_enable(RCC_USART1);

// UART TX on PA9 (GPIO_USART1_TX)
```

```
gpio_set_mode(GPIOA,
    GPIO_MODE_OUTPUT_50_MHZ,
    GPIO_CNF_OUTPUT_PUSHPULL,
    GPIO_USART1_TX);
```

I am repeating myself here because this is such an easy mistake to make. Yes, the code has configured PA9 for GPIO output. But this is *not* the same as the *peripheral* output. For that, you must configure it as an *alternate function* output (note argument three):

```
// UART TX on PA9 (GPIO_USART1_TX)
gpio_set_mode(GPIOA,
    GPIO_MODE_OUTPUT_50_MHZ,
    GPIO_CNF_OUTPUT_ALTFN_PUSHPULL,   // NOTE!!
    GPIO_USART1_TX);
```

This is what causes the peripheral to be connected to the GPIO pin and disconnects the regular GPIO function. Get this wrong, and you can be driven to madness. The code will *look correct* but will be laughing behind your back. Burn that into your consciousness early.

Alternate Function Fail

Your code performs some initialization for a peripheral to use a GPIO output, and you even use the correct GPIO_CNF_OUTPUT_ALTFN_OPENDRAIN macro for the CAN bus, but still no joy. Where is the bug?

```
rcc_peripheral_enable_clock(&RCC_APB1ENR,RCC_APB1ENR_CAN1EN);
rcc_periph_clock_enable(RCC_GPIOB);
gpio_set_mode(GPIOB,
    GPIO_MODE_OUTPUT_50_MHZ,
    GPIO_CNF_OUTPUT_ALTFN_OPENDRAIN,
    GPIO_CAN_PB_TX);
gpio_set_mode(GPIOB,
    GPIO_MODE_INPUT,
    GPIO_CNF_INPUT_FLOAT,
    GPIO_CAN_PB_RX);
gpio_primary_remap(
```

```
 AFIO_MAPR_SWJ_CFG_JTAG_OFF_SW_OFF,
 AFIO_MAPR_CAN1_REMAP_PORTB); // CAN_RX=PB8, CAN_TX=PB9
```

Give yourself a pat on the back if you said that the AFIO clock needs to be enabled:

```
rcc_periph_clock_enable(RCC_AFIO);
```

If you omit this call, the GPIO *remapping* won't function. It needs a clock. An omission like this can be insidious.

Peripheral Fail

This should be obvious, but peripherals need their own clocks enabled. For CAN bus, it required this call:

```
rcc_peripheral_enable_clock(&RCC_APB1ENR,RCC_APB1ENR_CAN1EN);
```

For other peripherals like the UART, the call may be simpler:

```
rcc_periph_clock_enable(RCC_USART1);
```

Obviously, if the peripheral's clock is disabled, as it is after reset, then it will act like a dead piece of silicon.

ISR FreeRTOS Crash

FreeRTOS has a rule about what can and can't be called from within an ISR. While a task may call xQueueSend() anytime, an ISR must use the xQueueSendFromISR() function (note the FromISR on the end of the function name). The reasons may vary by platform, but ISRs generally operate under very strict conditions.

Interrupts are asynchronous in nature, so any function call is suspect if it is not known to be reentrant. FreeRTOS takes special measures to make certain that the called function is safe when you use the correct name. Break this rule, and you may experience sporadic fails.

Stack Overflow

Unfortunately, stack sizes must be determined upfront when creating a task. For example:

```
xTaskCreate(monitor_task,"monitor",350,NULL,1,NULL);
```

Argument three in the call allocates 350 *words* of storage for that task's stack (each word is four bytes in length). Function `xTaskCreate()` allocates the stack from the heap. If the stack size is insufficient, memory corruption will follow, with unpredictable results.

An improvement would be to check for this condition and do something about it. FreeRTOS provides three ways to address this. This is determined by the `configCHECK_FOR_STACK_OVERFLOW` macro as defined in your `FreeRTOSConfig.h` file:

1. `configCHECK_FOR_STACK_OVERFLOW` is defined as 0. FreeRTOS will not check for overflow; this is the most efficient for operation.

2. `configCHECK_FOR_STACK_OVERFLOW` is defined as 1 so that FreeRTOS will perform a quick stack check. Less efficient than approach 1, but more efficient than 3.

3. `configCHECK_FOR_STACK_OVERFLOW` is defined as 2 so that FreeRTOS will perform a more thorough stack check. This is the least efficient operation.

When the macro is defined as non-zero, you must supply a function to be called when the stack has overflowed:

```
void
vApplicationStackOverflowHook(
  xTaskHandle *pxTask,
  signed portCHAR *pcTaskName
) {
    // do something, perhaps
    // flash an LED
}
```

When the stack overflow is detected, the hook function is called. Some memory corruption is likely to have already occurred by the time this hook is called, so it is best to signal it using the simplest of methods, like turning on an LED or flashing it so many times.

Estimating Stack Size

Estimating the stack size required can be difficult for functions that call into library routines, especially third-party ones. So, how do you confirm how much space is needed? FreeRTOS provides a function that helps:

```
#include "FreeRTOS.h"
#include "task.h"

// Returns # of words:
UBaseType_t uxTaskGetStackHighWaterMark(TaskHandle_t task);
```

The FreeRTOS documentation doesn't state what it returns, but the return value is in *words*. The function, when given the handle of a task, will return the number of unused stack words. If the task was created with 350 words of stack and used a maximum of 100 words so far, then the return value will be 250. In other words, the closer the return value is to zero, the more likely it is that the task is will overflow its stack.

The FreeRTOS documentation states that the function call can be costly and thus should only be used in debugging. But be careful even then because stack usage can vary with usage patterns. Even so, it is better than nothing when attempting to arrive at an estimate.

When a Debugger Doesn't Help

There are times when a debugger is impractical. When debugging device drivers, for example, there may be interrupts and timeouts involved where stepping through the code is not going to help. In this situation, you may want to collect clues like where it crashes or what the last successfully processed event was. In many of these difficult situations, having access to an LED or a GPIO can provide insights.

Within an ISR, you don't have the luxury of sending a message to an LCD display or a serial message to a terminal. Instead, you need to find simple ways to convey events, like activating LEDs. If you have a DSO (digital storage scope) or logic analyzer, emitting signals on multiple GPIOs can be very informative, especially when determining how much time is spent within an ISR.

In more extreme cases, you may need to set aside a trace buffer that your ISR can populate. Then, using GDB, you can interrupt the execution of the STM32 and examine that trace buffer.

Push/Pull or Open Drain

Some problems are centered on the use of the GPIO output. For example, many forum posts claim that the hardware slave select doesn't work for SPI. It *does* work, however, if you use *open-drain* configuration and a pull-up resistor. While this might be surprising, it does make sense when you consider that the STM32 supports multi-master mode SPI.

Any multi-mastered bus element must use a pull-up resistor because if one controller is holding the bus signal high, another MCU would have to fight in order to pull the same signal low. The pull-up resistor allows the signal to go high when no controller is active on the signal. It also allows any bus participant to pull the signal low without a fight.

This highlights another issue. When reading through STMicroelectronics datasheets, it helps to keep an eye out for the *fine print* and *footnotes*. A lot of tricky stuff lurks there.

Peripheral Defects

In rare cases, you may encounter peripheral behavior that is incorrect. The STM32 peripherals are complex silicon-state machines, and they sometimes have deficiencies in certain situations or in certain configurations. Search for and download the "STM32F1 Errata Sheet" PDF file for insight into what can go wrong. Usually a work-around is provided.

Reading the errata, you may notice that many of the problems pertain to debugging. This is a head's up that not everything you might see in a remote GDB session is representative of reality. Remote debugging is very useful but can run into difficulties in special situations.

Resources

Most of your time will likely be spent getting the STM32 peripherals to work the way you want them to. The more advanced your application is, the more likely it is that you will spend time working through peripheral issues.

The very best source of information about the peripherals is contained in STMicroelectronics' "reference manual" RM0008. At a minimum, you'll want to download this PDF and have it available for working through difficult issues. But this is not the only resource you want.

Search for and download the "STM32F103x8 STM32F103xB PDF" document. The one very important table contained in that document is table 5, "Medium-density STM32F103xx pin definitions." You might not be concerned about the pin definitions, but you'll find it a gold mine for summarizing what each GPIO pin is able to become with the correct configuration. To use Table 5, look down the column for LQFP48 for the STM32F103C8T6. Following down the column, you will find pin numbers. Pin 11, for example, is GPIO PA1 after reset and is configurable to be one of the following:

- USART2_RTS

- ADC12_IN1

- TIM2_CH2

And it is not 5-volt tolerant.

All of this is essential information that seems to belong in the reference manual, but isn't found there.

The section titled "Electrical Characteristics" will be of interest to those who are looking for estimates of power consumption. For example, Table 17, "Typical current consumption in Run mode, code with data processing running from Flash," indicates that the MCU running at 72 MHz will consume about 27 mA with all of the peripherals disabled. Several other tables and charts of this nature are available in that document.

libopencm3

Even though libopencm3 has an API wiki, I find myself needing answers that are not supplied or not obvious. I suspect that you will experience the same when developing new applications. Questions like these occur:

- When can I combine different values in a function call argument?

- When must they be supplied in separate calls?

These are two sides of the same question. First, here is the direct link to the API wiki pages:

```
http://libopencm3.org/docs/latest/html
```

Along the left side, the documentation is split up by STM32 family member. For the STM32F103, you want to drill down on "STM32F1." In some cases, the details are spelled out. For example, the function

```
void gpio_set(uint32_t gpioport,uint16_t gpios);
```

is described with the following:

> *Set a Group of Pins Atomic.*
>
> *Set one or more pins of the given GPIO port to 1 in an atomic operation.*

This tells us clearly that you can combine multiple GPIO references using the C or (|) operator in argument two; for example:

```
gpio_set(GPIOB,GPIO5|GPIO5);
```

It probably goes without saying that you cannot combine values for `gpioport`. There are other types of calls like this one:

```
bool usart_get_flag(uint32_t usart,uint32_t flag);
```

The singular name "flag" and the description "USART Read a Status Flag" both indicate the singular. What happens if you combine flags? While this may not be a safe or normal thing to do, the only way to answer that is to look at the source code. At the bottom of the description, you will find a link "Definition at line 107 of file usart_common_f124.c." If you click on that, it brings up the source-file listing of the module containing the function. From there, you can search or scroll down to the function definition and see that it is defined as follows:

```
bool usart_get_flag(uint32_t usart, uint32_t flag)
{
    return ((USART_SR(usart) & flag) != 0);
}
```

This tells you a couple of things:

1. If you supply multiple flags, you only get a `bool` result (any of the flags may cause it to return `true`). This is not likely what you want.

2. It tells you how to obtain the status flags yourself by use of the macro `USART_SR(usart)`. You may need, however, to include another header file to make this available.

The point of this section is to make you aware that you need to read the libopencm3 API descriptions carefully. If the argument type is an `enum` type, that almost guarantees that you shouldn't combine arguments. If the argument type is a signed or unsigned integer, you might be able to combine. Check the documentation before you assume. Where you don't find those necessary answers, "use the source, Luke."

FreeRTOS Task Priorities

FreeRTOS provides multi-tasking with multiple priority levels. Be aware that the priority mechanism may not be what you expect. Task priorities are arranged so that level zero is the lowest priority. Level `configMAX_PRIORITIES-1` is the highest task priority. The macro `configMAX_PRIORITIES` is defined within `FreeRTOSConfig.h`.

The idle task has priority zero. It runs when no other task is ready to run. There are some FreeRTOS configurable options for what happens during idle, which you can read about in their manual. The default is to just spin the CPU until a higher-priority task changes to the Ready state.

The FreeRTOS scheduler is designed to give CPU to tasks that are in the Ready or Running state. If you have one or more tasks in the Ready or Running state at a higher priority, then *no lower-priority task will run*. This is different than Linux, for example, where the CPU is shared with lower-priority processes. Under FreeRTOS, lower-priority processes require that *all* of the higher-priority tasks be in one of the following states:

- *Suspended* by calls like `vTaskSuspend()`

- *Blocked* by a blocking call like `xTaskNotifyWait()`

This has consequences for tasks that wait for a peripheral event. If the driver within a task performs a busy loop, then CPU is not given up until the preemptive interrupt occurs. Even when the busy loop calls upon `taskYIELD()`, the CPU is given to the other ready task at the same priority in round-robin sequence. Again, the only way that a lower-priority task will gain the CPU is when *all* tasks at the higher priority are either suspended or blocked.

This requires an adjustment to your Linux/Unix way of thinking, where the CPU is shared with every process that is ready to run. If you want that, then in FreeRTOS you must run all of your tasks at the same priority level. All tasks at the same level are scheduled in a round-robin fashion.

The way that this problem manifests itself is that the lower-priority tasks appear to be frozen or hung. This is a clear sign that your priority scheme needs adjustment or that tasks are not being blocked/suspended as intended.

Scheduling Within libopencm3

The library libopencm3 was developed independently of FreeRTOS. Consequently, when a peripheral driver waits for a peripheral event, it often includes a busy loop. Let's look at one example of what I mean:

```
void usart_wait_send_ready(uint32_t usart)
{
    /* Wait until the data has been transferred into the shift register. */
    while ((USART_SR(usart) & USART_SR_TXE) == 0);
}
```

The usart_wait_send_ready() function is called prior to sending the data byte to the USART. But notice the while loop—it simply burns CPU waiting for the USART_SR_TXE flag to become true. The effect of this is that the calling task will expend its entire time slice before preemption gives the CPU to another task. This gets the job done but is suboptimal.

To make better use of the CPU, it would be better to have the task yield its time slice to another task so that other useful work can be done. Unfortunately, there are no hooks for this in libopencm3. This leaves you with the following choices:

1. Live with it (perhaps it's not critical for your application).

2. Copy the function into your code and add a taskYIELD() call.

3. Modify your copy of the libopencm3 library.

4. Implement hook functionality and submit it to the libopencm3 volunteers.

The easiest fix is the second approach. Copy the function's code into your own application and amend it slightly to call `taskYIELD()`:

```
void usart_wait_send_ready(uint32_t usart)
{
    /* Wait until the data has been transferred into the shift register. */
    while ((USART_SR(usart) & USART_SR_TXE) == 0)
        taskYIELD(); // Make FreeRTOS friendly
}
```

Summary

We have concentrated on the STM32F103C8T6 member of the STM32 family in this book. This has allowed us to concentrate on a fixed number of features. There are, of course, other family member devices with additional peripherals, like the DAC (digital-to-analog converter), to name only one. If you now have an appetite for more-advanced challenges, consider a STM32F407 family device, like the Discovery board. If you're on a student budget, there are other choices, like the Core407V, on eBay. The STM32F4 provides much more in the way of SRAM, flash, and peripherals than we have considered in this book. It also includes hardware floating point, which can be performance critical to some applications.

I hope that this book has left you well informed about the STM32 platform and has given you fun challenges to work through. In celebration of this, there will be *no* exercises in this chapter! Thank you for allowing me to be your guide.

APPENDIX A

Answers to Exercises

Chapter 4

1. What GPIO port does the built-in LED on the Blue Pill PCB use? Specify the libopencm3 macro name for the port.

   Answer: PORTC

2. What GPIO pin does the built-in LED on the Blue Pill PCB use? Specify the libopencm3 macro name.

   Answer: GPIO13

3. What level is required to turn the built-in LED on for the Blue Pill PCB?

   Answer: logic low (or zero volts)

4. What are two factors affecting the chosen loop count in a programmed delay in non-multitasking environments?
   Answer:

   a. The CPU clock rate

   b. Instruction execution time

5. Why are programmed delays not used in a multi-tasking environment?

   Answer: Because the timing of other tasks in your system will affect the elapsed time of your programmed delay.

6.   What three factors that affect instruction timing?

Answer:

a.   the chosen platform

b.   CPU clock rate

c.   execution context (running code in flash or SRAM)

7.   What are the three modes of an input GPIO port?

Answer:

a.   Analog

b.   Digital, floating

c.   Digital, pull up and pull down

8.   Do the weak pull-up and pull-down resistors participate in an analog input?

Answer: No

9.   When is the Schmitt trigger enabled for input ports?

Answer: GPIO or peripheral digital input

10.   Do the weak pull-up and pull-down resistors participate for output GPIO ports?

Answer: No. They only apply to inputs.

11.   When configuring a USART TX (transmit) output for push/pull operation, which specialization macro should be used?

Answer: `GPIO_CNF_OUTPUT_ALTFN_PUSHPULL`

12.   When configuring a pin for LED use, which GPIO mode macro is preferred for low EMI?

Answer: `GPIO_MODE_OUTPUT_2_MHZ` (higher-rate choices like `GPIO_MODE_OUTPUT_10_MHZ` use more current and generate additional EMI).

Chapter 5

Answer the following:

1. How many tasks are running in blinky2?

   Answer: 1

2. How many threads of control are operating in blinky2?

   Answer: 2 threads: main thread and task 1

3. What would happen to the blink rate of blinky2 if the value of configCPU_CLOCK_HZ were configured as 36000000?

   Answer: The blink rate would double because the FreeRTOS scheduler is expecting the CPU to be half as fast.

4. Where does task 1's stack come from?

   Answer: Task 1's stack is allocated from the heap.

5. Exactly when does task1() begin?

   Answer: when the function vTaskStartScheduler() is called

6. Why is a message queue needed?

   Answer: to safely communicate between different threads of control

7. Even though it uses an execution delay loop, why does it seem to work with a nearly 50 percent duty cycle?

   Answer: Because there is only one task executing, the timing remains fairly consistent.

8. How difficult is it to estimate how long the LED on PC13 is on for? Why?

   Answer: Difficult due to instruction timing, flash prefetch, and so on

9. Using a scope, measure the on and off times of PC13 (or count how many blinks per second and compute the inverse). How many milliseconds is the LED on for?

   Answer: 84 ms

10. If another task were added to this project that consumed most of the CPU, how would the blink rate be affected?

    Answer: The blink rate would slow considerably.

11. Add to the file main.c a task 2 that does nothing but execute __asm__("nop") in a loop. Create that task in main() prior to starting the scheduler. How did that impact the blink rate? Why?

    Answer: It slowed considerably because the second task was consuming CPU time away from the first task.

Chapter 6

1. What is the idle state of the TTL level of a USART signal?

   Answer: High (near 5 volts)

2. USART data is provided in a big or little endian sequence?

   Answer: little endian (least significant bit first)

3. What clock(s) must be enabled for UART use?

   Answer: RCC_GPIOx and RCC_USARTn

4. What does the abbreviation 8N1 stand for?

   Answer: 8 bits of data, no parity, and 1 stop bit.

5. What happens if you provide UART data to be sent if the device is not yet empty?

   Answer: Data is lost.

6. Can tasks be created before, after, or before and after
`vTaskStartScheduler()`?

Answer: Before and after

7. What is the minimum buffer size determined by for
`xQueueReceive()`?

Answer: The receiving buffer size must meet or exceed the item
size as was specified when the queue was created.

8. How do you specify that `xQueueSend()` should return immediately
if the queue is full?

Answer: Supply argument `xTicksToWait` with the value 0.

9. How do you specify that `xQueueReceive()` should block forever if
the queue is empty?

Answer: Supply argument `xTicksToWait` with the macro `portMAX_`
`DELAY`.

10. What happens to the task if `xQueueReceive()` finds the queue
empty and it must wait?

Answer: The task will yield to another task.

Chapter 7

1. What GPIO preparation is necessary before enabling the USB
peripheral?

Answer: The GPIOA clock must be enabled, but otherwise the
USB peripheral takes over PA11 and PA12 automatically.

2. What are the alternate GPIO configurations available for USB?

Answer: There are no alternate configurations for USB. Only PA11
and PA12 are used.

3. What libopencm3 routine must be called regularly to handle USB events?

    Answer: The routine `usbd_poll()` must be called frequently to handle events that require action.

Chapter 8

1. How many data lines are used by SPI in bidirectional links? What are their signal names?

    Answer: There are two data lines used by SPI bidirectional links: MOSI and MISO.

2. Where does the clock originate from?

    Answer: The clock (SCK) is always provided by the master of the SPI bus.

3. What voltage levels are used for SPI signaling?

    Answer: The voltage levels used are usually 5 volts or 3.3 volts, according to the system design requirements. The STM32 device will use 3.3 volts.

4. Why must a pull-up resistor be used for the STM32 /NSS line?

    Answer: A pull-up resistor for /NSS must be used because the STM32 MCU configures the output as an open-drain output, regardless of how it was initially configured. Without the pull-up resistor, the select line will never go high.

5. Why must a dummy value be sent in some SPI transactions?

    Answer: A dummy value is written to cause the master peripheral to emit the clock pulses necessary for the slave to send its data. The slave always depends upon the SPI master to provide the clock.

Chapter 9

1. In the structure typedef'ed as `s_overlay`, why are members defined as character pointers rather than `long int`?

    Answer: When the byte size is calculated, you need character pointers. If the type were `long int`, then the calculated size would be in words instead of bytes.

2. Why was the `xflash` memory region added to the linker script?

    Answer: The `xflash` region was created to hold all of the W25QXX flash code, which will not appear in the MCU's flash. Additionally, this code is loaded into the `xflash` at starting address of zero, whereas the MCU flash started at 0x08000000 instead.

3. What is the purpose of the overlay stub function?

    Answer: The stub function calls the overlay manager to make sure the required code is loaded into the overlay region in SRAM. Once the function pointer is known, it must then pass on the calling arguments and return values, if any.

4. In the Gnu declaration `__attribute__((noinline, section(". ov_fee")))`, what is the purpose of `noinline`? Why is it needed?

    Answer: The attribute `noinline` prevents the compiler from treating the function as "inline" code. This is especially important for small functions that the compiler may optimize.

5. Where does the declaration `__attribute((section("…"))` belong?

    Answer: The `__attribute__((section("…"))` declaration may only appear in the function prototype.

Chapter 10

1. What are the three possible interrupt events from the RTC?

    Answer: The three interrupt sources are RTC (tick), Alarm, and Overflow.

2. What is the purpose of the calls `taskENTER_CRITICAL_FROM_ISR` and `taskEXIT_CRITICAL_FROM_ISR`?

    Answer: The `taskENTER_CRITICAL_FROM_ISR()` and `taskEXIT_CRITICAL_FROM_ISR()` calls block other interrupts from occurring while performing a critical operation.

3. How many bits wide is the RTC counter?

    Answer: The RTC counter is 32 bits wide.

4. Which clock source continues when the STM32 is powered down?

    Answer: The LSE clock (32.768 kHz crystal oscillator), which continues to work even when the supply voltage is off, *provided* that the battery voltage V_{BAT} supply is maintained

5. Which is the most accurate clock source?

    Answer: The most accurate clock source is the HSE clock because it is controlled by an 8 MHz crystal oscillator, but only while power is maintained.

Chapter 11

1. What is the byte value sent when reading from slave address $21 (hexadecimal)?

    Answer: Hexadecimal address $21 is $42 when shifted left by one bit. A read operation requires a 1-bit in the least significant position, which results in a byte value of $43.

2. When the master requests a response from a non-existing slave device on the bus, how does the NAK get received?

    Answer: To ACK a response, the slave must pull the SDA line low. If there is no slave acknowledging, the pull-up resistor keeps the line high, causing the NAK to be received by default.

3. What is the advantage of the /INT line from the PCF8574?

    Answer: The /INT line allows the slave device to notify the controlling MCU directly if an input line changes state. Otherwise, the MCU would need to busy the I2C bus with read requests to see when the line changes.

4. What does quasi-bidirectional mean in the context of the PCF8574?

    Answer: To receive an input signal, the GPIO port needs to be set weakly high so that an input driver can pull it low. This effectively makes it an input or an output port. However, if the GPIO port is set low, it cannot be used for input. For this reason, it is considered quasi-bidirectional.

5. What is the difference between sourcing and sinking current?

    Answer: When current is *sourced*, it is controlled and flows from the positive side through a load connected to ground (negative). When sinking current, the load is attached to the positive rail and current is switched on and off at the ground end instead.

Chapter 12

1. For AFIO output pins, what GPIO configuration macros must be used?

    Answer: GPIO outputs must use `GPIO_CNF_OUTPUT_ALTFN_PUSHPULL` or `GPIO_CNF_OUTPUT_ALTFN_OPENDRAIN` macros or the pin will remain unconnected to the peripheral (it will act as a regular GPIO).

2. What clock must be enabled for AFIO changes?

   Answer: The RCC_AFIO clock must be enabled by `rcc_periph_clock_enable()`.

3. What GPIO configuration macros be used for input pins?

   Answer: Inputs require no special treatment other than having the AFIO peripheral clock and the GPIO clock enabled, and the peripheral's needing the input initialized.

4. What is the purpose of the OLED D/C input serve?

   Answer: The D/C input line (to the OLED) allows it to distinguish between OLED *command* bytes (when low) and OLED graphics *data* (when high).

Chapter 13

1. In the demo program, what DMA controller aspects had to be changed before starting the next transfer?

   Answer: The start address and length were changed after the DMA channel was disabled.

2. Does each DMA channel have its own ISR routine?

   Answer: Yes, each DMA channel has its own ISR.

3. In a memory-to-peripheral transfer, like the demo, where does the DMA request come from?

   Answer: The DMA request comes from the peripheral, except in a memory-to-memory transfer. In the demo program, the SPI transmit buffer empty flag signaled the need for a transfer.

4. In the demo program where SPI was used, what were the three conditions necessary before a DMA transfer could begin?

   Answer: In the demo program where SPI was used, the following were necessary to cause the DMA to begin:

  a. The DMA channel must be enabled.

  b. The DMA TX enable for SPI must be enabled.

  c. SPI must be enabled.

Chapter 14

1. How is the internal STM32 temperature represented?

    Answer: As a voltage

2. How does `GPIO_CNF_INPUT_ANALOG` differ from the value `GPIO_CNF_INPUT_PULL_UPDOWN` or `GPIO_CNF_INPUT_FLOAT`?

    Answer: The configuration value `GPIO_CNF_INPUT_ANALOG` allows a varying voltage to reach the ADC input. Otherwise, only a low or high signal would be sensed.

3. If PCLK has a frequency of 36 MHz, what would be the ADC clock rate be when configured with a prescale divisor of 4?

    Answer: The ADC clock would be 36 MHz ÷ 4, which is 9 MHz.

4. Name three configuration options that affect the total power consumed by ADC.

    Answer: Three factors that affect power consumption for ADC are:

    a. `adc_power_on(adc)` (and off)

    b. `adc_enable_temperature_sensor()` (and disable)

    c. `adc_start_conversion_direct(adc)`

5. Assuming that the ADC clock after the prescaler is 12 MHz, how long does the `ADC_SMPR_SMP_41DOT5CYC` configured sample take?

    Answer: $(41.5 + 12.5) \div 12\ \text{MHz} = 54 \div 12e6 = 4.5\ \mu s$.

Chapter 15

1. What is the advantage of an RC clock?

   Answer: An RC clock requires no external crystal (crystals are too large to include on an IC).

2. What is the disadvantage of an RC clock?

   Answer: An RC clock is prone to drift and jitter and is less stable. It is also less precise, leading to problems with generating baud rates and so forth.

3. What is the advantage of a crystal-derived clock?

   Answer: A crystal-controlled clock is stable and can match external hardware. This makes it more ideal for generating baud rates and so forth.

4. What is the PLL used for?

   Answer: The PLL is used to multiply a clock to a rate higher than its input clock.

5. What does AHB stand for?

   Answer: AHB stands for AMBA High-performance Bus.

6. Why must the GPIO PA8 be configured with `GPIO_CNF_OUTPUT_ALTFN_PUSHPULL`?

   Answer: Without the `ALTFN` in the macro name, the GPIO would remain disconnected and otherwise be a normal GPIO having nothing to do with MCO output.

Chapter 16

1. In a RC Servo signal, what is the period of the signal?

   Answer: The period of a PWM signal is the time between the start of the pulse and the start of the next.

2.   Why is the timer input clock frequency 72 Mhz on the Blue Pill STM32F103C8T6? Why isn't it 36 MHz?

Answer: The input frequency to the timer is 72 MHz (for the Blue Pill STM32) because when the AHB1 prescaler is not equal to one, the timer frequency is the AHB1 bus frequency doubled.

3.   What is changed in the timer to effect a change in the pulse width?

Answer: The value of the output compare register

Chapter 17

1.   Why does the timer have a digital filter available on its inputs?

Answer: The digital filter eliminates false triggering from random noise pulses.

2.   When does the timer reset in PWM input mode?

Answer: As configured in the demo of Chapter 17, the counter resets after the capture 1 event occurs.

3.   Where does the IC2 input signal come from in PWM input mode?

Answer: In PWM input mode, the IC2 input comes from input channel 1.

Chapter 19

1.   How many FIFO's are supported by the STM32F103 CAN peripheral?

Answer: There are two FIFOs in the CAN peripheral (FIFO 0 and FIFO 1).

2.   How many filter banks are supported by the CAN peripheral?

Answer: There are two filter banks in the CAN peripheral (banks 0 and 1).

3. When a pair of filters must be supplied, but only one is needed, what are two ways to accomplish this?

    Answer: You can supply one filter when a pair are required in one of two ways:

    a. Declare two identical filters (only one will trigger)

    b. Declare one good filter and one impossible filter.

4. What is the RTR flag and what is its purpose?

    Answer: The RTR (remote transmission request) flag is used to request a transmission when it is sent in the recessive state. The reply is always sent with the RTR flag in the dominant state.

Chapter 20

1. What does the make macro `BINARY` define?

    Answer: The `BINARY` macro defines the name of the application executable with the `.elf` suffix attached.

2. What is the purpose of the header file `FreeRTOSConfig.h`?

    Answer: The header file `FreeRTOSConfig.h` configures several aspects of the FreeRTOS system.

3. How do you add compiler option `-O3` only to the compile of module `speedy.c`?

    Answer: In the Makefile, add the following rule: `speedy.o: CFLAGS += -O3`

4. What is the main disadvantage of using `heap_1`?

    Answer: The main disadvantage of using `heap_1.c` in a FreeRTOS project is that the function `free()` is *not* supported. No dynamically allocated memory can be released and reused.

APPENDIX B

STM32F103C8T6 GPIO Pins

This appendix is provided for convenience. This information is derived from the STMicroelectronics PDF document that can be downloaded by googling "STM32F103x8 STM32F103xB datasheet." The information here is extracted from Table 5, on page 28, for just the STM32F103C8T6, which is an LQFP48 device.

Pin	Name	Type	I/O Level	After Reset	Default	Remap
1	V_{BAT}	S	-	V_{BAT}	-	-
2	PC13-TAMPER-RTC	I/O	-	PC13	TAMPER-RTC	-
3	PC14-OSC32_IN	I/O	-	PC14	OSC32-IN	-
4	PC15-OSC32_OUT	I/O	-	PC15	OSC32-OUT	-
5	OSC_IN	I	-	OSC_IN	-	PD0
6	OSC_OUT	0	-	OSC_OUT	-	PD1
7	NRST	I/O	-	NRST	-	-
8	*VSSA*	S	-	*VSSA*	-	-
9	*VDDA*	S	-	*VDDA*	-	-

Pin	Name	Type	I/O Level	After Reset	Default	Remap
10	PA0-WKUP	I/O	-	PA0	WKUP	-
					USART2_CTS	
					ADC12_IN0	
					TIM2_CH1_ETR	
11	PA1	I/O	-	PA1	USART2_RTS	-
					ADC12_IN1	
					TIM2_CH2	
12	PA2	I/O	-	PA2	USART2_TX	-
					ADC12_IN2	
					TIM2_CH3	
13	PA3	I/O	-	PA3	USART2_RX	-
					ADC12_IN3	
					TIM2_CH4	

Pin	Name	Type	I/O Level	After Reset	Default	Remap
14	PA4	I/O	-	PA4	SPI1_NSS	-
					USART2_CK	
					ADC12_IN4	
15	PA5	I/O	-	PA5	SPI1_SCK	-
					ADC12_IN5	
16	PA6	I/O	-	PA6	SPI1_MISO	TIM1_BKIN
					ADC12_IN6	
					TIM3_CH1	
17	PA7	I/O	-	PA7	SPI1_MOSI	TIM1_CH1N
					ADC12_IN7	
					TIM3_CH2	
18	PB0	I/O	-	PB0	ADC12_IN8	TIM1_CH2N
					TIM3_CH3	
19	PB1	I/O	-	PB1	ADC12_IN9	TIM1_CH3N
					TIM3_CH4	
20	PB2	I/O	FT	PB2/BOOT1	-	-

Pin	Name	Type	I/O Level	After Reset	Default	Remap
21	PB10	I/O	FT	PB10	I2C2_SCL	TIM2_CH3
					USART3_TX	
22	PB11	I/O	FT	PB11	I2C2_SDA	TIM2_CH
					USART3_RX	
23	VSS_{1}	S	-	VSS_{1}	-	-
24	VDD_{1}	S	-	VDD_{1}	-	-
25	PB12	I/O	FT	PB12	SPI2_NSS	-
					I2C2_SMBAI	
					USART3_CK	
					TIM1_BKIN	
26	PB13	I/O	FT	PB13	SPI2_SCK	-
					USART3_CTS	
					TIM1_CH1N	
27	PB14	I/O	FT	PB14	SPI2_MISO	-
					USART3_RTS	
					TIM1_CH2N	

Pin	Name	Type	I/O Level	After Reset	Default	Remap
28	PB15	I/O	FT	PB15	SPI2_MOSI	-
					TIM1_CH3N	
29	PA8	I/O	FT	PA8	USART1_CK	-
					TIM1_CH1	
					MCO	
30	PA9	I/O	FT	PA9	USART1_TX	-
					TIM1_CH2	
31	PA10	I/O	FT	PA10	USART1_RX	-
					TIM1_CH3	
32	PA11	I/O	FT	PA11	USART1_CTS	-
					CANRX	
					USBDM	
					TIM1_CH4	

Pin	Name	Type	I/O Level	After Reset	Default	Remap
33	PA12	I/O	FT	PA12	USART1_RTS	-
					CANTX	
					USBDP	
					TIM1_ETR	
34	PA13	I/O	FT	JTMS/SWDIO	-	PA13
35	VSS_{2}	S	-	VSS_{2}	-	-
36	VDD_{2}	S	-	VDD_{2}	-	-
37	PA14	I/O	FT	JTCK/SWCLK	-	PA14
38	PA15	I/O	FT	JTDI	-	TIM2_CH1_ETR
						PA15
						SPI1_NSS
39	PB3	I/O	FT	JTDO	-	TIM2_CH2
						PB3
						TRACESWO
						SPI1_SCK

Pin	Name	Type	I/O Level	After Reset	Default	Remap
40	PB4	I/O	FT	JNTRST	-	TIM3_CH1
						PB4
						SPI1_MISO
41	PB5	I/O	FT	PB5	I2C1_SMBAI	TIM3_CH2
						SPI1_MOSI
42	PB6	I/O	FT	PB6	I2C1_SCL	USART1_TX
					TIM4_CH1	
43	PB7	I/O	FT	PB7	I2C1_SDA	USART1_RX
					TIM4_CH2	
44	BOOT0	I	-	BOOT0	-	-
45	PB8	I/O	FT	PB8	TIM4_CH3	I2C1_SCL
						CANRX
46	PB9	I/O	FT	PB9	TIM4_CH4	I2C1_SDA
						CANTX
47	VSS_3	S	-	VSS_3	-	-
48	VDD_3	S	-	VDD_3	-	-

Legend

Symbol	Description
I	Input
O	Output
S	Supply
FT	5-volt tolerant

Index

P, Q

R