# Discovering the STM32 Microcontroller


Geoffrey Brown

# Discovering the STM32 Microcontroller

Geoffrey Brown


June 5, 2016



# Contents

## List of exercises

# Acknowledgment

I have had a lot of help from various people in the Indiana University School of Informatics in developing these materials. Most notably, Caleb Hess developed the protoboard that we use in our lab, and he, along with Bryce Himebaugh made significant contributions to the development of the various experiments. Tracey Theriault provided many of the photographs.

I am grateful to ST Microelectronics for the many donations that allowed us to develop this laboratory. I particularly wish to thank Andrew Dostie who always responded quickly to any request that I made.

# Foreword

This book is intended as a hands-on manual for learning how to design systems using the STM32 F1 family of micro-controllers. It was written to support a junior-level computer science course at Indiana University. The focus of this book is on developing code to utilize the various peripherals available in STM32 F1 micro-controllers and in particular the STM32VL Discovery board. Because there are other fine sources of information on the Cortex-M3, which is the core processor for the STM32 F1 micro-controllers, we do not examine this core in detail; an excellent reference is "The Definitive Guide to the ARM CORTEX-M3." [5]

This book is not exhaustive, but rather provides a single "trail" to learning about programming STM32 micro controller built around a series of laboratory exercises. A key design decision was to utilize readily available off-the-shelf hardware models for all the experiments discussed.

I would be happy to make available to any instructor the other materials developed for teaching C335 (Computer Structures) at Indiana University; however, copyright restrictions limit my ability to make them broadly available.

Geoffrey Brown
Indiana University

# Chapter 1

# Getting Started

The last few years has seen a renaissance of hobbyists and inventors building custom electronic devices. These systems utilize off-the-shelf components and modules whose development has been fueled by a technological explosion of integrated sensors and actuators that incorporate much of the analog electronics which previously presented a barrier to system development by non-engineers. Micro-controllers with custom firmware provide the glue to bind sophisticated off-the-shelf modules into complex custom systems. This book provides a series of tutorials aimed at teaching the embedded programming and hardware interfacing skills needed to use the STM32 family of micro-controllers in developing electronic devices. The book is aimed at readers with 'C' programming experience, but no prior experience with embedded systems.

The STM32 family of micro-controllers, based upon the ARM Cortex-M3 core, provides a foundation for building a vast range of embedded systems from simple battery powered dongles to complex real-time systems such as helicopter autopilots. This component family includes dozens of distinct configurations providing wide-ranging choices in memory sizes, available peripherals, performance, and power. The components are sufficiently inexpensive in small quantities – a few dollars for the least complex devices – to justify their use for most low-volume applications. Indeed, the low-end "Value Line" components are comparable in cost to the ATmega parts which are used for the popular Arduino development boards yet offer significantly greater performance and more powerful peripherals. Furthermore, the peripherals used are shared across many family members (for example, the USART modules are common to all STM32 F1 components) and are supported by a single firmware library. Thus, learning how to program one member of the STM32 F1 family

enables programming them all. [1]

Unfortunately, power and flexibility are achieved at a cost – software development for the STM32 family can be extremely challenging for the uninitiated with a vast array of documentation and software libraries to wade through. For example, RM0041, the reference manual for large value-line STM32 F1 devices, is 675 pages and does not even cover the Cortex-M3 processor core ! Fortunately, it is not necessary to read this book to get started with developing software for the STM32, although it is an important reference. In addition, a beginner is faced with many tool-chain choices. [2] In contrast, the Arduino platform offers a simple application library and a single tool-chain which is accessible to relatively inexperienced programmers. For many simple systems this offers a quick path to prototype. However, simplicity has its own costs – the Arduino software platform isn't well suited to managing concurrent activities in a complex real-time system and, for software interacting with external devices, is dependent upon libraries developed outside the Arduino programming model using tools and techniques similar to those required for the STM32. Furthermore, the Arduino platform doesn't provide debugging capability which severely limits the development of more complex systems. Again, debugging requires breaking outside the confines of the Arduino platform. Finally, the Arduino environment does not support a real-time operating system (RTOS), which is essential when building more complex embedded systems.

For readers with prior 'C' programming experience, the STM32 family is a far better platform than the Arduino upon which to build micro-controller powered systems if the barriers to entry can be reduced. The objective of this book is to help embedded systems beginners get jump started with programming the STM32 family. I do assume basic competence with C programming in a Linux environment – readers with no programming experience are better served by starting with a platform like Arduino. I assume familiarity with a text editor; and experience writing, compiling, and debugging C programs. I do not assume significant familiarity with hardware – the small amount of "wiring" required in this book can easily be accomplished by a rank beginner.

The projects I describe in this book utilize a small number of read-

[1] There are currently five families of STM32 MCUs – STM32 F0, STM32 F1, STM32 L1, STM32 F2, and STM32 F4 supported by different, but structurally similar, firmware libraries. While these families share many peripherals, some care is needed when moving projects between these families. [18, 17, 16]

[2] A tool-chain includes a compiler, assembler, linker, debugger, and various tools for processing binary files.

ily available, inexpensive, off-the-shelf modules. These include the amazing STM32 VL Discovery board (a $10 board that includes both an STM32 F100 processor and a hardware debugger link), a small LCD display, a USB/UART bridge, a Wii Nunchuk, and speaker and microphone modules. With this small set of components we can explore three of the most important hardware interfaces – serial, SPI, and I2C – analog input and output interfaces, and the development of firmware utilizing both interrupts and DMA. All of the required building blocks are readily available through domestic suppliers as well as ebay vendors. I have chosen not to utilize a single, comprehensive, "evaluation board" as is commonly done with tutorials because I hope that the readers of this book will see that this basic collection of components along with the software techniques introduced provides the concepts necessary to adapt many other off-the-self components. Along the way I suggest other such modules and describe how to adapt the techniques introduced in this book to their use.

The development software used in this book is all open-source. Our primary resource is the GNU software development tool-chain including gcc, gas, objcopy, objdump, and the debugger gdb. I do not use an IDE such as eclipse. I find that most IDEs have a high startup cost although they can ultimately streamline the development process for large systems. IDEs also obscure the compilation process in a manner that makes it difficult to determine what is really happening, when my objective here is to lay bare the development process. While the reader is welcome to use an IDE, I offer no guidance on setting one up. One should not assume that open-source means lower quality – many commercial tool-chains for embedded systems utilize GNU software and a significant fraction of commercial software development is accomplished with GNU software. Finally, virtually every embedded processor is supported by the GNU software tool-chain. Learning to use this tool-chain on one processor literally opens wide the doors to embedded software development.

Firmware development differs significantly from application development because it is often exceedingly difficult to determine what is actually happening in code that interacts with a hardware peripheral simply through examining program state. Furthermore, in many situations it is impractical to halt program execution (e.g., through a debugger) because doing so would invalidate real-time behavior. For example, in developing code to interface with a Wii Nunchuk (one of the projects described in this book) I had difficulty tracking down a timing bug which related to how fast data was being "clocked" across the hardware interface. No amount of software debugging

could have helped isolate this problem – I had to have a way to see the hardware behavior. Similarly, when developing code to provide flow-control for a serial interface, I found my assumptions about how the specific USB/UART bridge I was communicating with were wrong. It was only through observing the hardware interface that I found this problem.

In this book I introduce a firmware development process that combines traditional software debugging (with GDB), with the use of a low-cost "logic analyzer" to allow the capture of real-time behavior at hardware interfaces.

## 1.1 Required Hardware

A list of the hardware required for the tutorials in this book is provided in Figure 1.1. The component list is organized by categories corresponding to the various interfaces covered by this book followed by the required prototyping materials and test equipment. In the remainder of this section, I describe each of these components and, where some options exist, key properties that must be satisfied. A few of these components require header pins to be soldered on. This is a fairly simple task that can be accomplished with even a very low cost pencil soldering iron. The amount of soldering required is minimal and I recommend borrowing the necessary equipment if possible. There are many soldering tutorials on the web.

The most expensive component required is a logic analyzer. While I use the Saleae Logic it may be too expensive for casual hobbyists ($150).[3] An alternative, OpenBench Logic Sniffer, is considerably cheaper ($50) and probably adequate. My choice was dictated by the needs of a teaching laboratory where equipment takes a terrific beating – the exposed electronics and pins of the Logic Sniffer are too vulnerable for such an environment. An Oscilloscope might be helpful for the audio interfaces, but is far from essential.

### STM32 VL Discovery

The key component used in the tutorials is the STM32 VL discovery board produced by STMicroelectronics (ST) and available from many electronics distributors for approximately $10. [4] This board, illustrated in Figure 1.2 includes a user configurable STM32 F100 micro-controller with 128 KB flash and 8 KB ram as well as an integrated hardware debugger interface based upon a dedicated USB connected STM32 F103. With appropriate software

[3] At the time of writing Saleae offers a discount to students and professors.
[4] http://www.st.com/internet/evalboard/product/250863.jsp

| Component | Supplier | cost |
|---|---|---|
| **Processor** | | |
| STM32 VL discovery | Mouser, Digikey, Future Electronics | $10 |
| **Asynchronous Serial** | | |
| USB/UART breakout | Sparkfun, Pololu, ebay | $7-$15 |
| **SPI** | | |
| EEPROM (25LC160) | Digikey, Mouser, others | $0.75 |
| LCD (ST7735) | ebay and adafruit | $16-$25 |
| Micro SD card (1-2G) | Various | $5 |
| **I2C** | | |
| Wii Nunchuk | ebay (clones), Amazon | $6-$12 |
| Nunchuk Adaptor | Sparkfun, Adafruit | $3 |
| **Time Based** | | |
| Hobby Servo (HS-55 micro) | ebay | $5 |
| Ultrasonic range finder (HC-SR04) | ebay | $4 |
| **Analog** | | |
| Potentiometer | Digikey, Mouser, ebay | $1 |
| Audio amplifier | Sparkfun (TPA2005D1) | $8 |
| Speaker | Sparkfun COM-10722 | $1 |
| Microphone Module | Sparkfun (BOB-09868 or BOB-09964) | $8-$10 |
| **Power Supply (optional)** | | |
| Step Down Regulator (2110) | Pololu | $15 |
| 9V Battery Holder | | |
| 9V Battery | | |
| **Prototyping Materials** | | |
| Solderless 700 point breadboard (2) | ebay | $6 |
| Jumper wires | ebay | $5-$10 |
| **Test Equipment** | | |
| Saleae Logic or Oscilloscope | Saleae optional for testing analog output | $150 |

Figure 1.1: Required Prototype Hardware and Suppliers

running on the host it is possible to connect to the STM32 F100 processor to download, execute, and debug user code. Furthermore, the hardware debug-

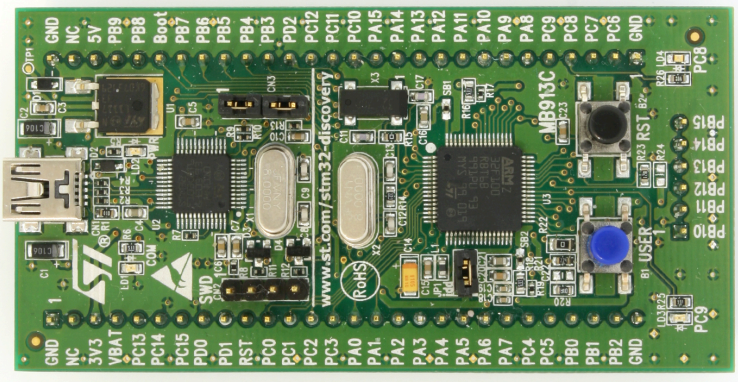



Figure 1.2: STM32 VL Discovery Board

ger interface is accessible through pin headers and can be used to debug any member of the STM32 family – effectively, ST are giving away a hardware debugger interface with a basic prototyping board. The STM32 VL Discovery board is distributed with complete documentation including schematics. [14].

In the photograph, there is a vertical white line slightly to the left of the midpoint. To the right of the line are the STM32 F100, crystal oscillators, two user accessible LEDs, a user accessible push-button and a reset push button. To the left is the hardware debugger interface including an STM32 F103, voltage regulator, and other components. The regulator converts the 5V supplied by the USB connection to 3.3V for the processors and also available at the board edge connectors. This regulator is capable of sourcing sufficient current to support the additional hardware used for the tutorials.

All of the pins of the STM32 F100 are brought out to well labeled headers – as we shall see the pin labels directly correspond to the logical names used throughout the STM32 documentation rather than the physical pins associated with the particular part/package used. This use of logical names is consistent across the family and greatly simplifies the task of designing portable software.

The STM32 F100 is a member of the value line STM32 processors and executes are a relatively slow (for Cortex-M3 processors) 24Mhz, yet provides far more computation and I/O horsepower than is required for the tutorials described in this book. Furthermore, all of the peripherals provided by the STM32 F100 are common to the other members of the STM32 family and, the code developed on this component is completely portable across the micro-

controller family.

### Asynchronous Serial

One of the most useful techniques for debugging software is to print messages to a terminal. The STM32 micro-controllers provide the necessary capability for serial communications through USART (universal synchronous asynchronous receiver transmitter) devices, but not the physical connection necessary to communicate with a host computer. For the tutorials we utilize a common USB/UART bridge. The most common of these are meant as serial port replacements for PCs and are unsuitable for our purposes because they include voltage level converters to satisfy the RS-232 specification. Instead we require a device which provides more direct access to the pins of the USB/UART bridge device.

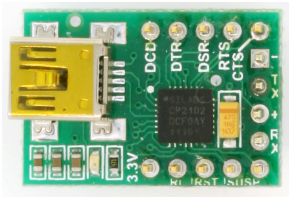



Figure 1.3: Pololu CP2102 Breakout Board

An example of such a device, shown in Figure 1.3 is the Pololu cp2102 breakout board. An alternative is the Sparkfun FT232RL breakout board (BOB-00718) which utilizes the FTDI FT232RL bridge chip. I purchased a cp2102 board on ebay which was cheap and works well. While a board with either bridge device will be fine, it is important to note that not all such boards are suitable. The most common cp2102 boards, which have a six pin header, do not provide access the the hardware flow control pins that are essential for reliable high speed connection. An important tutorial in this book covers the implementation of a reliable high-speed serial interface. You should look at the pin-out for any such board to ensure at least the following signals are available – `rx, tx, rts, cts`.

Asynchronous serial interfaces are used on many commonly available modules including GPS (global positioning system) receivers, GSM cellular modems, and bluetooth wireless interfaces.


Figure 1.4: EEPROM in PDIP Package

## SPI

The simplest of the two synchronous serial interfaces that we examine in this book is SPI. The key modules we consider are a color LCD display and an SD flash memory card. As these represent relatively complex uses of the SPI interface, we first discuss a simpler device – a serial EEPROM (electrically erasable programmable memory). Many embedded systems use these for persistent storage and it is relatively simple to develop the code necessary to access them.

There are many EEPROMs available with similar, although not identical interfaces. I recommend beginning with the Microchip 25LC160 in a PDIP package (see Figure 1.4). Other packages can be challenging to use in a basic prototyping environment. EEPROMs with different storage densities frequently require slightly different communications protocols.

The second SPI device we consider is a display – we use an inexpensive color TFT (thin film transistor) module that includes a micro SD card adaptor slot. While I used the one illustrated in Figure 1.1, an equivalent module is available from Adafruit. The most important constraint is that the examples in this book assume that the display controller is an ST7735 with a SPI interface. We do use the SD card adaptor, although it is possible to find alternative adaptors from Sparkfun and others.

The display is 128x160 pixel full color display similar to those used on devices like ipods and digital cameras. The colors are quite bright and can easily display images with good fidelity. One significant limitation to SPI based displays is communication bandwidth – for high speed graphics it would be advisable to use a display with a parallel interface. Although the value line component on the discovery board does not provide a built-in peripheral to support parallel interfaces, many other STM32 components do.

Finally you will need an SD memory card in the range 1G-2G along with an adaptor to program the card with a desktop computer. The speed

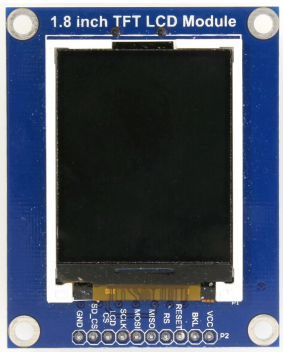



Figure 1.5: Color Display Module

and brand are not critical. The recommended TFT module includes an SD flash memory card slot.

### I2C

Figure 1.6: Wii Nunchuk

The second synchronous serial interface we study is I2C. To illustrate the use of the I2C bus we use the Wii Nunchuk (Figure 1.6). This was developed and used for the Wii video console, but has been re-purposed by hobbyists. It contains an ST LIS3L02AL 3-axis accelerometer, a 2-axis analog joy-stick,

and two buttons all of which can be polled over the I2C bus. These are widely available in both genuine and clone form. I should note that there appear to be some subtle differences between the various clones that may impact software development. The specific problem is a difference in initialization sequences and data encoding.


Figure 1.7: Wii Nunchuk Adaptor

The connector on the Nunchuk is proprietary to Wii and I have not found a source for the mating connector. There are simple adaptor boards available that work well for the purposes of these tutorials. These are available from several sources; the Sparkfun version is illustrated in Figure 1.7.

### Time Based

Hardware timers are key components of most micro-controllers. In addition to being used to measure the passage of time – for example, providing an alarm at regular intervals – timers are used to both generate and decode complex pulse trains. A common use is the generation of a pulse-width modulated signal for motor speed control. The STM32 timers are quite sophisticated and support complex time generation and measurement. We demonstrate how timers can be used to set the position of common hobby servos (Figure 1.8) and to measure time-of-flight for an ultrasonic range sensor (Figure 1.9). The ultrasonic range sensor we use is known generically as an HC-SR04 and is available from multiple suppliers – I obtained one from an ebay vendor. Virtually any small hobby servo will work, however, because of the power limitations

of USB it is desirable to use a "micro" servo for the experiments described in this book.

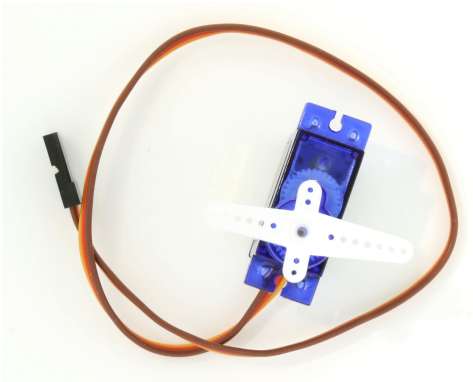


Figure 1.8: Servo

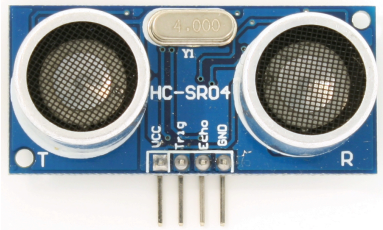



Figure 1.9: Ultrasonic Sensor

## Analog

The final interface that we consider is analog – both in (analog to digital) and out (digital to analog). A digital to analog converter (DAC) translates a digital value into a voltage. To illustrate this capability we use a DAC to drive a small speaker through an amplifier (Figure 1.11). The particular experiment, reading audio files off an SD memory card and playing then through a speaker, requires the use of multiple interfaces as well as timers and DMA.

To illustrate the use of analog to digital conversion, we use a small potentiometer (Figure 1.10) to provide a variable input voltage and a microphone (Figure 1.12) to provide an analog signal.

Figure 1.10: Common Potentiometer


Figure 1.11: Speaker and Amplifier


Figure 1.12: Microphone

## Power Supply

In our laboratory we utilize USB power for most experiments. However, if it is necessary to build a battery powered project then all that is needed is a voltage regulator (converter) between the desired battery voltage and 5V. The STM32 VL Discovery includes a linear regulator to convert 5V to 3.3V. I have used a simple step-down converter step-down converter – Figure 1.13 illustrates one available from Pololu – to convert the output of a 9V battery to 5V. With such a converter and battery, all of the experiments described in this book can be made portable.

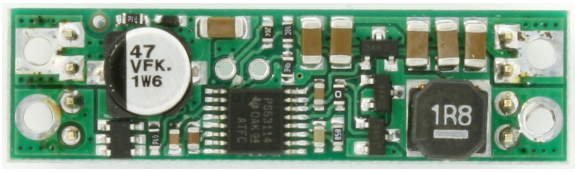



Figure 1.13: Power Supply

## Prototyping Materials

### Need pictures

In order to provide a platform for wiring the various components together, I recommend purchasing two 700-tie solder less bread boards along with a number of breadboard jumper wires in both female-female and male-male configuration. All of these are available on ebay at extremely competitive prices.

## Test Equipment

The Saleae Logic logic analyzer is illustrated in Figure 1.14. This device provides a simple 8-channel logic analyzer capable of capturing digital data at 10-20 MHz which is sufficiently fast to debug the basic serial protocols utilized by these tutorials. While the hardware itself is quite simple – even primitive – the software provided is very sophisticated. Most importantly, it has the capability of analyzing several communication protocols and displaying the resulting data in a meaningful manner. Figure 1.15 demonstrates the display of serial data – in this case "hello world" (you may need to zoom in your pdf viewer to see the details).

When developing software in an embedded environment, the most likely scenario when testing a new hardware interface is ... nothing happens. Unless things work perfectly, it is difficult to know where to begin looking for problems. With a logic analyzer, one can capture and visualize any data that is being transmitted. For example, when working on software to drive a serial port, it is possible to determine whether anything is being transmitted, and if so, what. This becomes especially important where the embedded processor is communicating with an external device (e.g. a Wii Nunchuk) – where every command requires a transmitting and receiving a specific binary sequence. A logic analyzer provides the key to observing the actual communication events (if any !).

Figure 1.14: Saleae Logic

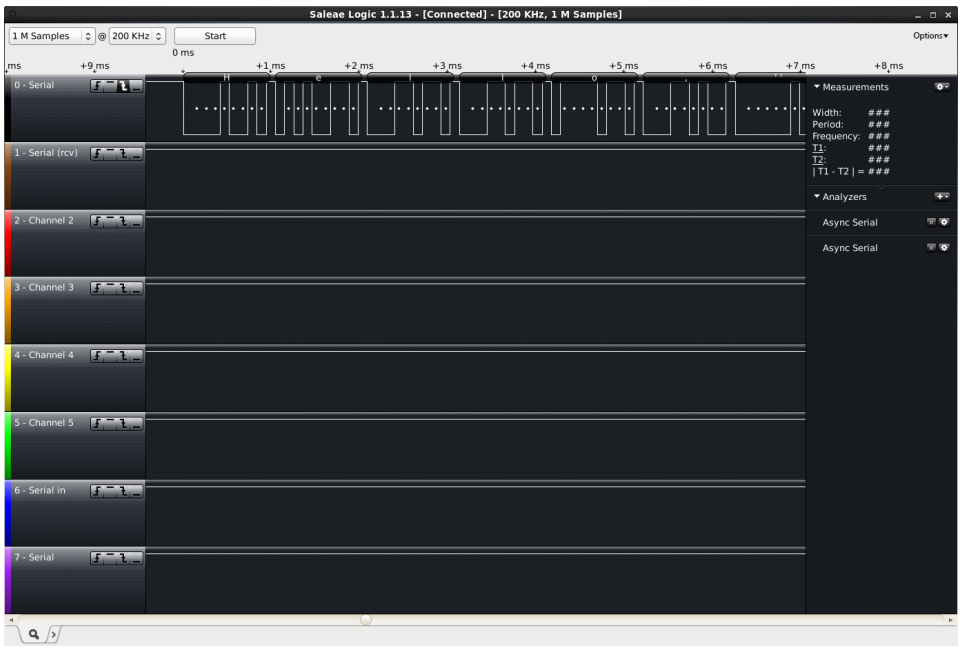



Figure 1.15: Saleae Logic Software

## 1.2 Software Installation

The software development process described in this book utilizes the firmware libraries distributed by STMicroelectronics, which provide low-level

access to all of the peripherals of the STM32 family. While these libraries are relatively complicated, this book will provide a road map to their use as well some initial shortcuts. The advantages to the using these firmware libraries are that they abstract much of the bit-level detail required to program the STM32, they are relatively mature and have been thoroughly tested, and they enable the development of application code that is portable across the STM32 family. In contrast, we have examined the sample code distributed with the NXP LPC13xx Cortex-M3 processors and found it to be incomplete and in a relatively immature state.

### GNU Tool chain

The software development for this book was performed using the GNU embedded development tools including gcc, gas, gdb, and gld. We have successfully used two different distributions of these tools. In a linux environment we use the Sourcery (a subsidiary of Mentor Graphics) CodeBench Lite Edition for ARM (EABI). These may be obtained at `https://sourcery.mentor.com/sgpp/lite/arm/portal/subscription?@template=lite`. I recommend using the GNU/Linux installer. The site includes PDF documentation for the GNU tool chain along with a "getting started" document providing detailed installation instructions.

Adding the following to your Linux bash initialization will make access simpler

```
export PATH=path-to/codesourcery/bin:$PATH
```

On OS X systems (Macs) we use the yagarto (`www.yagarto.de`) distribution of the GNU toolchain. There is a simple installer available for download.

### STM32 Firmware Library

The STM32 parts are well supported by a the ST Standard Peripheral Library [5] which provides firmware to support all of the peripherals on the various STM32 parts. This library, while easy to install, can be quite challenging to use. There are many separate modules (one for each peripheral) as well as large numbers of definitions and functions for each module. Furthermore, compiling with these modules requires appropriate compiler flags as well as

[5] `http://www.st.com/web/en/catalog/tools/PF257890`

a few external files (a configuration file, and a small amount of code). The approach taken in this documentation is to provide a basic build environment (makefiles, configuration file, etc.) which can be easily extended as we explore the various peripherals. Rather than attempt to fully describe this peripheral library, I present modules as needed and then only the functions/definitions we require.

### Code Template

While the firmware provided by STMicroelectronics provides a solid foundation for software development with the STM32 family, it can be difficult to get started. Unfortunately, the examples distributed with the STM32 VL Discovery board are deeply interwoven with the commercial windows-based IDEs available for STM32 code development and are challenging to extract and use in a Linux environment. I have created a small template example which uses standard Linux make files and in which all aspects of the build process are exposed to the user.

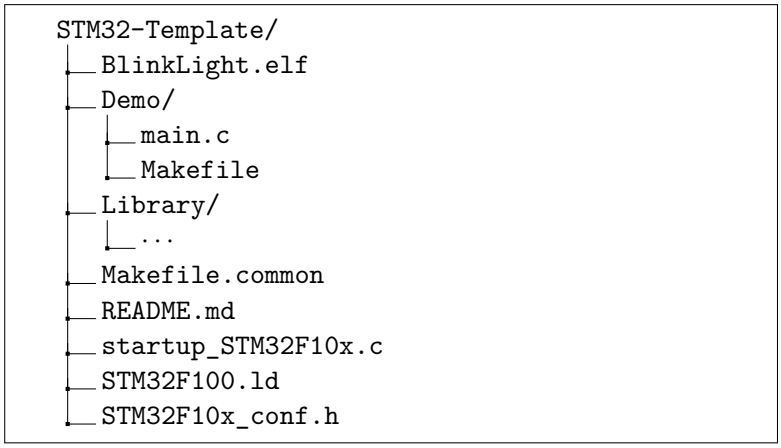



Figure 1.16: STM32VL Template

This template can be downloaded as follows:

```
git clone git://github.com/geoffreymbrown/STM32-Template.git
```

The template directory (illustrated in Figure 1.16) consists of part specific startup code, a part specific linker script, a common makefile, and a

header file required by the standard peripheral library. A subdirectory contains the code and example specific makefile. The directory includes a working binary for the STM32 VL Discovery. The Demo program is discussed further in Chapter 3.

### GDB Server

In order to download and debug code on the STM32 VL Discovery board we can exploit the built-in USB debugger interface called stlink which communicates with the STM32 on-chip debug module. The stlink interface can be used both for the processor on the Discovery board and, by setting jumper appropriately, for off-board processors. ST also sells a stand-alone version of this debugger interface. Sadly, the stlink interface is only supported on Windows and ST has not publicly released the interface specification. It is widely known that the stlink interface is implemented using the USB Mass Storage device class and it is further known that this particular implementation is incompatible with the OS X and Linux kernel drivers. Nevertheless, the interface has been sufficiently reverse-engineered that a very usable gdb server running on Linux or OS X is available for download:

```
git clone git://github.com/texane/stlink.git
```

The README file describes the installation process. The STM32VL Discovery board utilizes the STLINKv1 protocol which is somewhat problematic in either case because of the manner in which it interacts with the OS Kernel. Because of the kernel issues, it is important to follow the directions provided. In the case of OS X, there is also a "mac os x driver" which must be built and installed.

To execute the gdb server, plug in an STM32 VL discovery board. Check to see if "/dev/stlink" exists and then execute:

```
st-util -1
```

Note: earlier versions of st-util need a different startup sequence

```
st-util 4242 /dev/stlink
```

To download the blinking light example, start an instance of arm-none-eabi-gdb in a separate window and execute the following

```
arm-none-eabi-gdb BlinkingLights.elf
(gdb) target extended-remote :4242
(gdb) load
(gdb) continue
```

This will download the program to flash and begin execution.

GDB can also be used to set breakpoints and watchpoints.

## 1.3 Key References

There are an overwhelming number of documents pertaining the the STM32 family of Cortex-M3 MCUs. The following list includes the key documents referred to in this book. Most of these are available on-line from `www.st.com`. The Cortex-M3 technical reference is available from `www.arm.com`.

**RM0041** Reference manual for STM32F100x Advanced ARM-based 32-bit MCUs [20]. This document provides reference information on all of the peripheral used in the STM32 value line processors including the processor used on the STM32 VL Discovery board.

**PM0056** STM32F10xx/20xx/21xx/L1xxx [19]. ST reference for programming the Cortex-M3 core. Include the execution model and instruction set, and core peripherals (e.g. the interrupt controller).

**Cortex-M3** ARM Cortex-M3 (revision r1p1) Technical Reference Manual. The definitive source for information pertaining to the Cortex-M3 [1].

**Data Sheet** Low & Medium-density Value Line STM32 data sheet [15]. Provides pin information – especially the mapping between GPIO names and alternative functions. There are data sheets for a number of STM32 family MCUs – this one applies to the MCU on the STM32 VL discovery board.

**UM0919** User Manual STM32 Value Line Discovery [14]. Provides detailed information, including circuit diagrams, for the STM32 VL Discovery board.

# Chapter 2

# Introduction to the STM32 F1

The STM32 F1xx micro-controllers are based upon the ARM Cortex-M3 core. The Cortex-M3 is also the basis for micro-controllers from a number of other manufacturers including TI, NXP, Toshiba, and Atmel. Sharing a common core means that the software development tools including compiler and debugger are common across a wide range of micro-controllers. The Cortex-M3 differs from previous generations of ARM processors by defining a number of key peripherals as part of the core architecture including interrupt controller, system timer, and debug and trace hardware (including external interfaces). This additional level of integration means that system software such as real-time operating systems and hardware development tools such as debugger interfaces can be common across the family of processors. The various Cortex-M3 based micro-controller families differ significantly in terms of hardware peripherals and memory – the STM32 family peripherals are completely different architecturally from the NXP family peripherals even where they have similar functionality. In this chapter we introduce key aspects of the Cortex-M3 core and of the STM32 F1xx micro-controllers.

A block diagram of the STM32F100 processor used on the value line discovery board is illustrated in Figure 2.1. The Cortex-M3 CPU is shown in the upper left corner. The value line components have a maximum frequency of 24 MHz – other STM32 processors can support a 72 MHz clock. The bulk of the figure illustrates the peripherals and their interconnection. The discovery processor has 8K bytes of SRAM and 128K bytes of flash. There are two peripheral communication buses – APB2 and APB1 supporting a wide variety of peripherals.

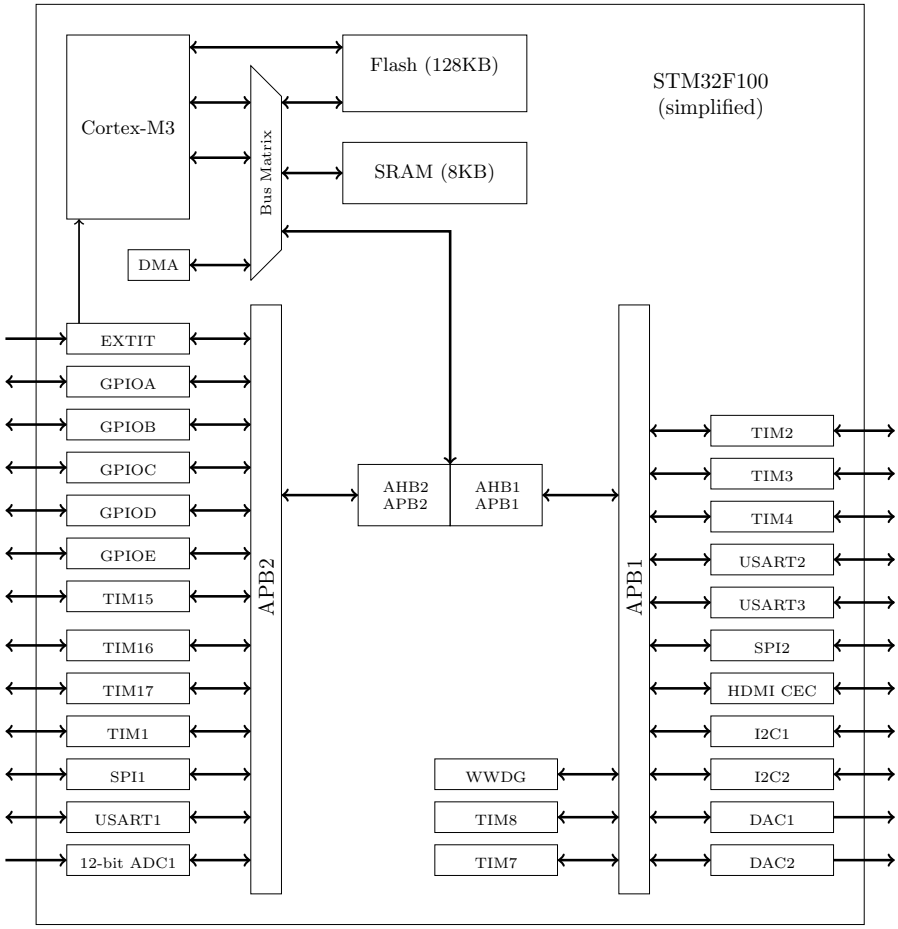



Figure 2.1: STM32 F100 Architecture

The Cortex-M3 core architecture consists of a 32-bit processor (CM3) with a small set of key peripherals – a simplified version of this core is illustrated in Figure 2.2. The CM3 core has a Harvard architecture meaning that it uses separate interfaces to fetch instructions (Inst) and (Data). This helps ensure the processor is not memory starved as it permits accessing data and instruction memories simultaneously. From the perspective of the CM3, everything looks like memory – it only differentiates between instruction fetches and data accesses. The interface between the Cortex-M3 and manufacturer

specific hardware is through three memory buses – ICode, DCode, and System – which are defined to access different regions of memory.

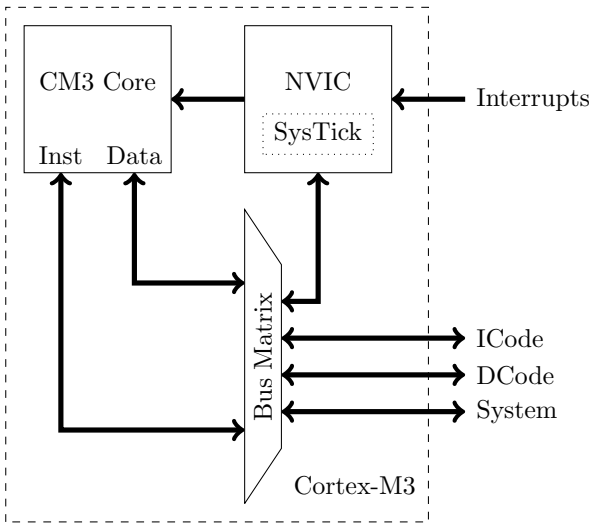



Figure 2.2: Simplified Cortex-M3 Core Architecture

The STM32, illustrated in Figure 2.3 connects the three buses defined by the Cortex-M3 through a micro-controller level bus matrix. In the STM32, the ICode bus connects the CM3 instruction interface to Flash Memory, the DCode bus connects to Flash memory for data fetch and the System bus provides read/write access to SRAM and the STM32 peripherals. The peripheral sub-system is supported by the AHB bus which is further divided into two sub-bus regions AHB1 and AHB2. The STM32 provides a sophisticated direct memory access (DMA) controller that supports direct transfer of data between peripherals and memory.

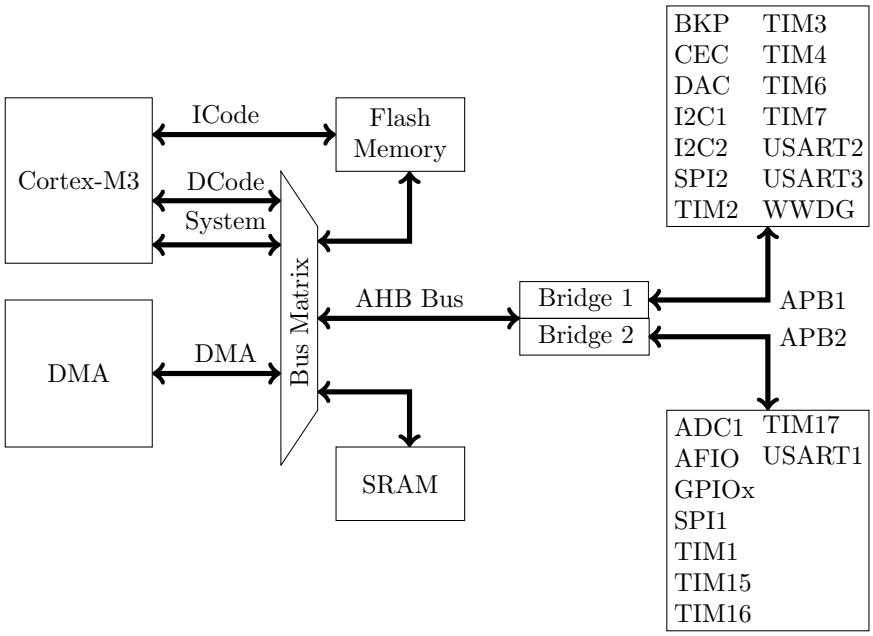



Figure 2.3: STM32 Medium Density Value-Line Bus Architecture

## 2.1 Cortex-M3

The CM3 processor implements the Thumb-2 instruction set which provides a large set of 16-bit instructions, enabling 2 instructions per memory fetch, along with a small set of 32-bit instructions to support more complex operations. The specific details of this instruction set are largely irrelevant for this book as we will be performing all our programming in C. However, there are a few key ideas which we discuss in the following.

As with all RISC processors, the Cortex-M3 is a load/store architecture with three basic types of instructions – register-to-register operations for processing data, memory operations which move data between memory and registers, and control flow operations enabling programming language control flow such as if and while statements and procedure calls. For example, suppose we define the following rather trivial C-procedure:

```
int counter;

int counterInc(void){
  return counter++;
}
```

The resulting (annotated) assembly language with corresponding machine code follows:

```
counterInc:
  0: f240 0300   movw    r3, #:lower16:counter  // r3 = &counter
  4: f2c0 0300   movt    r3, #:upper16:counter
  8: 6818        ldr     r0, [r3, #0]          // r0 = *r3
  a: 1c42        adds    r2, r0, #1            // r2 = r0 + 1
  c: 601a        str     r2, [r3, #0]          // *r3 = r2
  e: 4740        bx      lr                    // return r0
```

Two 32-bit instructions (movw, movt) are used to load the lower/upper halves of the address of `counter` (known at link time, and hence 0 in the code listing). Then three 16-bit instructions load (ldr) the value of counter, increment (adds) the value, and write back (str) the updated value. Finally, the procedure returns the original counter.

It is not expected that the reader of this book understand the Cortex-M3 instruction set, or even this example in great detail. The key points are that the Cortex-M3 utilizes a mixture of 32-bit and 16-bit instructions (mostly the latter) and that the core interacts with memory solely through load and store instructions. While there are instructions that load/store groups of registers (in multiple cycles) there are no instructions that directly operate on memory locations.

The Cortex-M3 core has 16 user-visible registers (illustrated in Figure 2.4) – all processing takes place in these registers. Three of these registers have dedicated functions including the program counter (PC), which holds the address of the next instruction to execute, the link register (LR), which holds the address from which the current procedure was called, and "the" stack pointer (SP) which holds the address of the current stack top (as we shall discuss in Chapter 11, the CM3 supports multiple execution modes, each with their own private stack pointer). Separately illustrated is a processor status register (PSR) which is implicitly accessed by many instructions.

The Cortex-M3, like other ARM processors was designed to be programmed (almost) entirely in higher-level language such as C. One consequence is a well developed "procedure call standard" (often called an ABI or

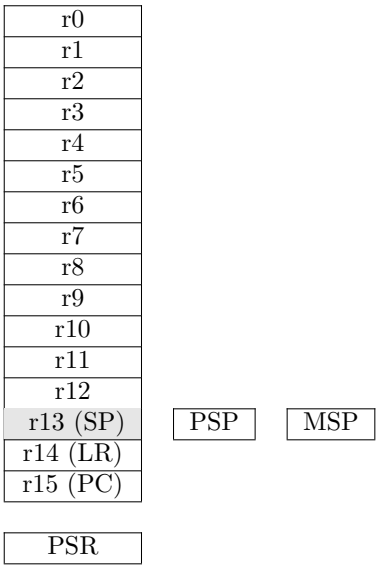



Figure 2.4: Processor Register Set

application binary interface) which dictates how registers are used. [2] This model explicitly assumes that the RAM for an executing program is divided into three regions as illustrated in Figure 2.5. The data in RAM are allocated during the link process and initialized by startup code at reset (see Chapter 3). The (optional) heap is managed at runtime by library code implementing functions such as the `malloc` and `free` which are part of the standard C library. The stack is managed at runtime by compiler generated code which generates per-procedure-call stack frames containing local variables and saved registers.

The Cortex-M3 has a "physical" address space of $2^{32}$ bytes. The ARM Cortex-M3 Technical Reference Manual defines how this address space is to be used. [1] This is (partially) illustrated in Figure 2.6. As mentioned, the "Code" region is accessed through the ICode (instructions) and DCode (constant data) buses. The SRAM and Peripheral areas are accessed through the System bus. The physical population of these regions is implementation dependent. For example, the STM32 processors have 8K–1M flash memory based at address (0x08000000). [1] The STM32F100 processor on the Discovery board has 8K of SRAM based at address 0x20000000. Not shown on this address map are the internal Cortex-M3 peripherals such as the NVIC which is located starting at

[1] This memory is "aliased" to 0x00000000 at boot time.

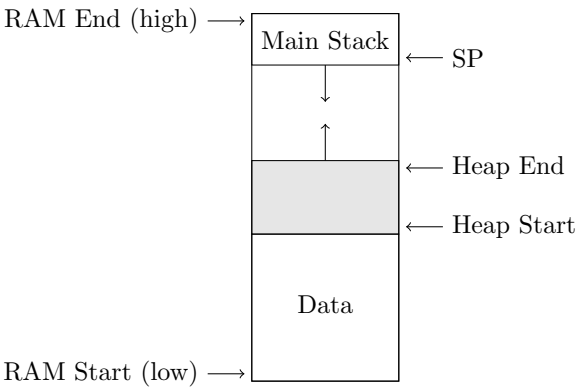



Figure 2.5: Program Memory Model

address 0xE000E000; these are defined in the Cortex-M3 reference manual. [1] We discuss the NVIC further in Chapter 11.

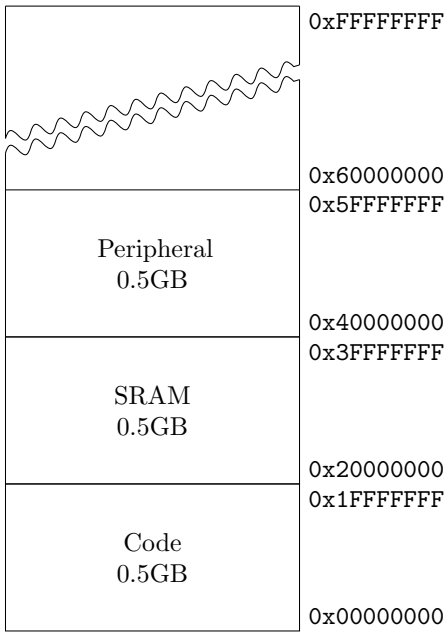



Figure 2.6: Cortex-M3 Memory Address Space

As mentioned, the Cortex-M3 core includes a vectored interrupt controller (NVIC) (see Chapter 11 for more details). The NVIC is a programmable device that sits between the CM3 core and the micro-controller. The Cortex-M3 uses a prioritized vectored interrupt model – the vector table is defined to reside starting at memory location 0. The first 16 entries in this table are defined for all Cortex-M3 implementations while the remainder, up to 240, are implementation specific; for example the STM32F100 devices define 60 additional vectors. The NVIC supports dynamic redefinition of priorities with up to 256 priority levels – the STM32 supports only 16 priority levels. Two entries in the vector table are especially important: address 0 contains the address of the initial stack pointer and address 4 contains the address of the "reset handler" to be executed at boot time.

The NVIC also provides key system control registers including the System Timer (SysTick) that provides a regular timer interrupt. Provision for a built-in timer across the Cortex-M3 family has the significant advantage of making operating system code highly portable – all operating systems need at least one core timer for time-slicing. The registers used to control the NVIC are defined to reside at address `0xE000E000` and are defined by the Cortex-M3 specification. These registers are accessed with the system bus.

## 2.2 STM32 F1

The STM32 is a family of micro-controllers. The STM32 F1xx micro-controllers are based upon the Cortex-M3 and include the STM32F100 value-line micro-controller used on the discovery board considered in this book. The STM32 L1 series is derived from the STM32 F1 series but with reduced power consumption. The STM32 F2 series is also based upon the Cortex-M3 but has an enhanced set of peripherals and a faster processor core. Many of the peripherals of the STM32 F1 series are forward compatible, but not all. The STM32 F4 series of processors use the Cortex-M4 core which is a significant enhancement of the Cortex-M3. Finally, there is a new STM32 family – the STM32 F0 based upon the Cortex-M0. Each of these families – STM32F0, STM32 F1, STM32 L1. STM32 F2, and STM32 F4 are supported by different firmware libraries. While there is significant overlap between the families and their peripherals, there are also important differences. In this book we focus on the STM32 F1 family.

As illustrated in Figure 2.3, the STM32 F1 micro-controllers are based upon the Cortex-M3 core with a set of peripherals distributed across three buses – AHB and its two sub-buses APB1 and APB2. These peripherals are

controlled by the core with load and store instructions that access memory-mapped registers. The peripherals can "interrupt" the core to request attention through peripheral specific interrupt requests routed through the NVIC. Finally, data transfers between peripherals and memory can be automated using DMA. In Chapter 4 we discuss basic peripheral configuration, in Chapter 11 we show how interrupts can be used to build effective software, and in Chapter 12 we show how to use DMA to improve performance and allow processing to proceed in parallel with data transfer.

Throughout this book we utilize the ST Standard Peripheral Library for the STM32 F10xx processors. It is helpful to understand the layout of this software library. Figure 2.7 provides a simplified view of the directory structure. The library consists of two major sub-directories – STM32F10x_StdPeriph_Driver and CMSIS. CMSIS stands for "Cortex Micro-controller Software Interface Standard" and provides the common low-level software required for all ARM Cortex parts. For example, the core_cm3.* files provide access to the interrupt controller, the system tick timer, and the debug and trace modules. The STM32F10x_StdPeriph_Driver directory provides roughly one module (23 in all) for each of the peripherals available in the STM32 F10x family. In the figure, I have included modules for general purpose I/O (GPIO), I2C, SPI, and serial IO (USART). Throughout this book I will introduce the modules as necessary.

There are additional directories distributed with the firmware libraries that provide sample code which are not illustrated. The supplied figure provides the paths to all of the key components required to build the tutorials in this book.

The STM32 F1 has a sophisticated clock system. There are two primary external sources of timing – HSE and LSE. The HSE signal is derived from an 8MHz crystal or other resonator, and the LSE signal is derived from a 32.768 kHz crystal. Internally, the HSE is multiplied in frequency through the use of a PLL; the output of this, SYSCLK is used to derive (by division) various on-chip time sources include clocks for the ABP1 and APB2 peripherals as well as for the various programmable timers. The LSE is used to manage a low-power real-time clock. The STM32F100 micro-controllers can support a maximum SYSCLK frequency of 24MHz while the other STM32 F1xx micro-controllers support a SYSCLK frequency of 72MHz. Fortunately, most of the code required to manage these clocks is provided in the standard peripheral library module (`system_stm32f10x.[ch]`) which provides an initialization function – `SystemInit(void)` to be called at startup. This module also exports a variable `SystenCoreClock` which contains the SYSCLK frequency; this simplifies

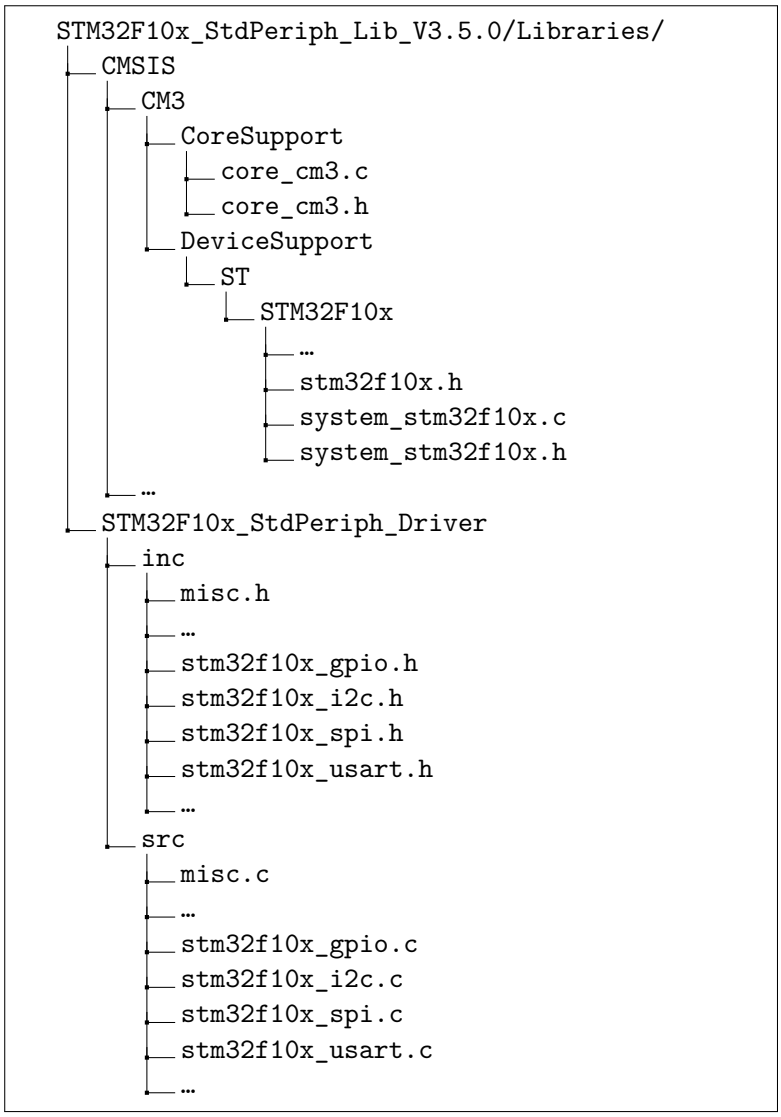



Figure 2.7: ST Standard Peripheral Library

the task of developing code that is portable across the STM32F1 family.

The STM32 F1 micro-controllers a variety of peripherals – not all of which are supported by the STM32F100 parts. The following peripherals are considered extensively in this book.

**ADC** Analog to digital converter – Chapter 14.

**DAC** Digital to analog converter – Chapter 13.

**GPIO** General Purpose I/O – Chapter 4.

**I2C** I2C bus – Chapter 9.

**SPI** SPI bus – Chapter 6.

**TIM** Timers (various) – Chapter 10.

**USART** Universal synchronous asynchronous receiver transmitter – Chapter 5.

The following peripherals are not considered in this book.

**CAN** Controller area network. Not supported by STM32F100

**CEC** Consumer electronics control.

**CRC** Cyclic redundancy check calculation unit.

**ETH** Ethernet interface. Not supported by the STM32F100

**FSMC** Flexible static memory controller. Not supported by medium density STMF100.

**PWR** Power control (sleep and low power mode).

**RTC** Real time clock.

**IWDG** Independent watchdog.

**USB** Universal serial bus. Not supported by the STM32F100

**WWDG** Windowing watchdog

As mentioned previously all of the peripherals are "memory-mapped" which means that the core interacts with the peripheral hardware by reading and writing peripheral "registers" using load and store instructions. [2] All of

[2] The terminology can be confusing – from the perspective of the CM3 core, peripheral registers are just dedicated memory locations.

the various peripheral registers are documented in the various STM32 reference manuals ([20, 21]). The documentation include bit-level definitions of the various registers and text to help interpret those bits. The actual physical addresses are also found in the reference manuals.

The following table provides the address for a subset of the peripherals that we consider in this book. Notice that all of these fall in the area of the Cortex-M3 address space defined for peripherals.

| | |
|---|---|
| `0x40013800 - 0x40013BFF` | USART1 |
| `0x40013000 - 0x400133FF` | SPI1 |
| `0x40012C00 - 0x40012FFF` | TIM1 timer |
| `0x40012400 - 0x400127FF` | ADC1 |
| ... | ... |

Fortunately, it is not necessary for a programmer to look up all these values as they are defined in the library file `stm32f10x.h` as `USART1_BASE`, `SPI1_BASE`, `TIM1_BASE` `ADC1_BASE`, etc.

Typically, each peripheral will have control registers to configure the peripheral, status registers to determine the current peripheral status, and data registers to read data from and write data to the peripheral. Each GPIO port (GPIOA, GPIOB, etc.) has seven registers. Two are used to configure the sixteen port bits individually, two are used to read/write the sixteen port bits in parallel, two are used to set/reset the sixteen port bits individually, and one is used to implement a "locking sequence" that is intended to prevent rogue code from accidentally modifying the port configuration. This final feature can help minimize the possibility that software bugs lead to hardware failures; e.g, accidentally causing a short circuit.

In addition to providing the addresses of the peripherals, `stm32f10x.h` also provides C language level structures that can be used to access each peripherals. For example, the GPIO ports are defined by the following register structure.

```
typedef struct
{
  volatile uint32_t CRL;
  volatile uint32_t CRH;
  volatile uint32_t IDR;
  volatile uint32_t ODR;
  volatile uint32_t BSRR;
  volatile uint32_t BRR;
  volatile uint32_t LCKR;
} GPIO_TypeDef;
```

The register addresses of the various ports are defined in the library as (the following defines are from `stm32f10x.h`)

```
#define PERIPH_BASE          ((uint32_t)0x40000000)
#define APB2PERIPH_BASE      (PERIPH_BASE + 0x10000)
#define GPIOA_BASE           (APB2PERIPH_BASE + 0x0800)
#define GPIOA                ((GPIO_TypeDef *) GPIOA_BASE)
```

To read the 16 bits of GPIOA in parallel we might use the following code:

```
uint16_t GPIO_ReadInputData(GPIO_TypeDef* GPIOx) {
  return ((uint16_t)GPIOx->IDR);
}
```

The preceding example is somewhat misleading in its simplicity. Consider that to configure a GPIO pin requires writing two 2-bit fields at the correct location in correct configuration register. In general, the detail required can be excruciating.

Fortunately, the standard peripheral library provides modules for each peripheral that can greatly simplify this task. For example, the following is a subset of the procedures available for managing GPIO ports:

```
void GPIO_Init(GPIO_TypeDef* GPIOx,
               GPIO_InitTypeDef* GPIO_InitStruct);
uint8_t GPIO_ReadInputDataBit(GPIO_TypeDef* GPIOx,
                              uint16_t GPIO_Pin);
uint16_t GPIO_ReadInputData(GPIO_TypeDef* GPIOx);
uint8_t GPIO_ReadOutputDataBit(GPIO_TypeDef* GPIOx,
                               uint16_t GPIO_Pin);
uint16_t GPIO_ReadOutputData(GPIO_TypeDef* GPIOx);
void GPIO_SetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
void GPIO_ResetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
void GPIO_WriteBit(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin,
                   BitAction BitVal);
void GPIO_Write(GPIO_TypeDef* GPIOx, uint16_t PortVal);
```

The initialization function (`GPIO_Init`) provides an interface for configuring individual port bits. The remaining functions provide interfaces for reading and writing (also setting and resetting) both individual bits and the 16 port bits in parallel.

We use the standard peripheral library functions throughout this book.

There is a significant downside to using this library – the modules are huge. The GPIO module `stm32f10x_gpio.o` when compiled with parameter checking is 4K where a simple application might use a few 100 bytes of custom

code. Furthermore, the code can be slow – often multiple procedure calls are used by the library where none would be required by custom code. Nevertheless, the library offers a much faster path to correct prototype code. For prototype work, it's probably better to throw extra hardware (memory, clock rate) at a problem than sweat the details. For serious product development it may be wise to refine a design to reduce dependence on these libraries.

To get a sense of the cost of using the library consider the code in Figure 2.1 which configures PC8 and PC9 as outputs (to drive LEDs) and PA0 as an input (to read the push button). [3]. Similar library based code is presented as an exercise in Chapter 4. In Table 2.1 I compare the space requirements of two versions of this program with and without the use of the standard peripheral library. The first column (text) provides the size of "text segment" (code and data initializers), the data allocated in ram at startup is the sum of data (initialized data) and bss (zeroed data). The total memory requirements are provided in column text. The .elf files are the complete binaries. Excluding 256 bytes of preallocated runtime stack (bss), the library version is nearly 3 times as large. Unlike the original which did minimum system initialization, I included two common startup files for both versions. Also, the standard peripheral library has extensive parameter checking which I disabled for this comparison.

| Code With Libraries | | | | |
|---|---|---|---|---|
| text | data | bss | dec | filename |
| 200 | 0 | 0 | 200 | main.o |
| 576 | 0 | 0 | 576 | startup_stm32f10x.o |
| 832 | 0 | 0 | 832 | stm32f10x_gpio.o |
| 1112 | 20 | 0 | 1132 | stm32f10x_rcc.o |
| 484 | 20 | 0 | 504 | system_stm32f10x.o |
| 3204 | 40 | 256 | 3500 | blinky2-lib.elf |
| Code Without Libraries | | | | |
| text | data | bss | dec | filename |
| 136 | 0 | 0 | 136 | main.o |
| 576 | 0 | 0 | 576 | startup_stm32f10x.o |
| 484 | 20 | 0 | 504 | system_stm32f10x.o |
| 1196 | 20 | 256 | 1472 | blinky2.elf |

Table 2.1: Code Size With and Without Standard Libraries

[3] This is an excerpt of blinky.c by Paul Robson

```
#include <stm32f10x.h>
int main(void){
    int n = 0;
    int button;

    /* Enable the GPIOA (bit 2) and GPIOC (bit 4) */
    /* See 6.3.7 in stm32f100x reference manual   */

    RCC->APB2ENR |= 0x10 | 0x04;

    /* Set GPIOC Pin 8 and Pin 9 to outputs       */
    /* 7.2.2 in stm32f100x reference manual       */

    GPIOC->CRH = 0x11;

    /* Set GPIOA Pin 0 to input floating          */
    /* 7.2.1 in stm32f100x reference manual       */

    GPIOA->CRL = 0x04;

    while(1){
      delay();
         // Read the button - the button pulls down PA0 to logic 0
      button = ((GPIOA->IDR & 0x1) == 0);
      n++;

      /* see 7.2.5 in stm32f100x reference manual */

      if (n & 1) {
        GPIOC->BSRR = 1<<8 ;
        } else {
        GPIOC->BSRR = 1<<24;
        }
      if ((n & 4) && button) {
        GPIOC->BSRR = 1<<9 ;
        } else {
        GPIOC->BSRR = 1<<25;
        }
    }
}

void delay(void){
  int i = 100000;/* About 1/4 second delay */
  while (i-- > 0)
    asm("nop");
}
```

Listing 2.1: Programming without Standard Peripheral Library

# Chapter 3

# Skeleton Program

In this chapter I discuss the process of creating, compiling, loading, executing, and debugging a program with the STM32 VL Discovery board and Sourcery tools. For desktop machines, the standard first example is the "hello world" program:

```
#include <stdio.h>
main() {
  printf("hello world\n");
}
```

which can be compiled and executed in a single step

```
$ gcc -o hello hello.c ; ./hello
hello world
```

This simple program hides an enormous amount of complexity ranging from the automatic inclusion of the standard libraries, to linking in startup code, to interacting with the world through the shell. In the embedded world, much of that complexity is visible to the programmer and hence it is necessary to understand quite a bit more about the execution environment for even the simplest program (and "hello world" is not a simple program).

In the embedded world, the simplest C program is one which does not require any standard libraries and does not interact with the world:

```
main {
}
```

However, this is a little too pared down for our purposes. Instead, we structure this chapter around a program that has some data and which runs forever:

```
int i;
int off = 5;

void inc(void){
  i += off;
}

int main(void){
  while (1) {
    inc();
  }
}
```

While we cannot directly observe this program when it executes, we can attach a debugger and control its execution through breakpoints and watchpoints. Notice that this program has two global variables (`i` and `off`) one of which is initialized to zero and the other has a non-zero initizializer. Furthermore the program has a single procedure other than main and repeatedly calls this procedure.

Before we can execute the program there are a number of hurdles we must overcome. First, we must compile the program into a binary format suitable for loading onto the discovery board. Second, we must load this binary into the flash memory. Finally, in order to observe the program, we must intereact with the discovery board through a debugger (GDB). While we use GDB as a loader as well as a debugger, in general the last two steps may involve separate tools.

### Demo Program

The process of compiling a program for embedded processors such as the STM32 can involve quite a few details such as processor specific compilation flags, paths to the compilation tools, etc. Generally the best approach is to build a "make" script to guide the process. Rather than diving in at this level, you should download the code template as described in Section 1.2 which contains the necessary scripts. The layout of this directory is illustrated in Figure 1.16.

In order to build this example on your system, you will need to modify two constants in the file `Makefile.common` – `TOOLROOT` which should point

to the bin directory of your Sourcery installation and `LIBROOT` which should point to your installation of the STM32 standard periperal library.

To compile this program, change directories to the Demo directory and execute "make". This should create a file called `Demo.ELF` which contains the compiled binary.

To download and execute this binary we will need two programs – gdb (arm-none-eabi-gdb), which is part of the Sourcery distribution, and `st-util`, which provides a gdb server that communicates with the stlink debugging stub on the discovery board through a USB connection. We described how to install st-util in Section 1.2. I will assume you have installed and tested the connection. You should open two terminal windows. In one, execute:

```
st-util -1
```

Note: earlier versions of st-util need a different startup sequence

```
st-util 4242 /dev/stlink
```

which starts a gdb server listening at port 4242. You should see an output such as the following:

```
Chip ID is 00000420, Core ID is  1ba01477.
KARL - should read back as 0x03, not 60 02 00 00
Listening at *:4242...
```

In the other terminal window, execute (lines starting "(gdb)" are within the debugger):

```
arm-none-eabi-gdb Demo.elf
(gdb) target extended-remote :4242
(gdb) load
(gdb) break main
(gdb) break inc
(gdb) continue
```

The "target" command should connect to the gdb server at port 4242; the load command downloads the executable to the STM32 flash memory. The next two commands set breakpoints at `main` and `inc` procedures, and the continue command executes until the next breakpoint occurs. You can then repeatedly execute and examine the value of `i`:

```
(gdb) print i
(gdb) continue
...
```

### Exercise 3.1 GDB on STM32

Experiment with GDB to test various commands such as the following:

1. Print current register values (e.g. `print /x $sp` displays the stack pointer in hex.
2. Try setting a watchpoint on `i`.
3. Try using a breakpoint command that prints `i` and continues just before `main` calls `inc`

## Make Scripts

While downloading and executing a binary is comparatively easy, the process of building a binary is not. The compilation tools require non-obvious options, the STM32 firmware libraries require various definitions, and the generation of an executable from binaries requires a dedicated "linker script". Furthermore, "main.c" is not in itself a complete program – there are always steps necessary to initialize variables and set up the execution environment. In the Unix world, every C program is linked with "crt0.o" to perform this initialization. In the embedded world, additional initialization is necessary to set up hardware environment. In this section I discuss the build process and in the next, the function performed by the STM32 startup code (which is included with the firmware libraries).

The make files included with the demo program are split into two parts – `Makefile.common` does the heavy lifting and is reusable for other projects while `Demo/Makefile` is project specific. Indeed the only function of the project specific makefile is to define the required object files and their dependencies. The Makefile for the demo project is illustrated in Listing 3.1. This can be modified for other projects by adding additional objects and modifying compilation flags. The variable `TEMPLATEROOT` should be modified to point to the template directory.

```
TEMPLATEROOT = ..

# compilation flags for gdb

CFLAGS  = -O1 -g
ASFLAGS = -g

# object files

OBJS=  $(STARTUP) main.o

# include common make file

include $(TEMPLATEROOT)/Makefile.common
```

Listing 3.1: Demo Makefile

Most of `Makefile.common` defines paths to the tools and libraries. The only notable parts of this file are the processor specific defines and the compilation flags. The processor specific definitions include the linker script `LDSCRIPT` that informs the linker of the correct linking script to use – we will discuss this file briefly in the next section. A processor type define `PTYPE` that controls conditional compilation of the firmware libraries, and two startup files – one generic (`system_stm32f10x.o`) and one specific to the STM32 Value Line processors (`startup_stm32f10x.o`). The STM32 libraries are designed to support multiple members of the STM32 family by requiring various compile time switches. The processor on the STM32 VL Discovery board is "medium density value line" part – this is reflected in the compile-time definition `STM32F10X_MD_VL`.

```
PTYPE = STM32F10X_MD_VL
LDSCRIPT = $(TEMPLATEROOT)/stm32f100.ld
STARTUP= startup_stm32f10x.o system_stm32f10x.o

# Compilation Flags

FULLASSERT = -DUSE_FULL_ASSERT

LDFLAGS+= -T$(LDSCRIPT) -mthumb -mcpu=cortex-m3
CFLAGS+= -mcpu=cortex-m3 -mthumb
CFLAGS+= -I$(TEMPLATEROOT) -I$(DEVICE) -I$(CORE) -I$(PERIPH)/inc -I.
CFLAGS+= -D$(PTYPE) -DUSE_STDPERIPH_DRIVER $(FULLASSERT)
```

The flags `-mcpu=cortex-m3 -mthumb` inform gcc about the core proces-

sor. `-DUSE_STDPERIPH_DRIVER -DUSE_FULL_ASSERT` affect the compilation of firmware library code.

### STM32 Memory Model and Boot Sequence

The memory of the STM32 processors consists of two major areas – flash memory (effectively read-only) begins at address 0x08000000 while static ram (read/write) memory begins at address 0x20000000. The size of these areas is processor specific. When a program is executing, the machine code (generally) resides in flash and the mutable state (variables and the run-time stack) resides in static ram (SRAM). In addition, the first portion of flash memory, starting at 0x08000000, contains a vector table consisting of pointers to the various exception handlers. The most important of these are the address of the reset handler (stored at 0x08000004) which is executed whenever the processor is reset, and the initial stack pointer value (stored at 0x08000000).

This memory structure is reflected in the linker script fragment illustrated in Figure 3. The script begins by defining the code entry point (`Reset_Handler`) and the two memory regions – flash and ram. It then places the named sections from the object files being linked into appropriate locations in these two memory regions. From the perspective of an executable, there are three relevant sections – ".text" which is always placed in flash, ".data" and ".bss" which are always allocated space in the ram region. The constants required to initialize .data at runtime are placed in flash as well for the startup code to copy. Notice further that the linker script defines key labels `_etext, _sidata, ...` that are referenced by the startup code in order to initialize the ram.

The GNU linker is instructed to place the data section in FLASH – specifically "at" the location of `_sidata`, but links data references to locations in RAM by the following code fragment:

```
.data  : AT ( _sidata )
{
   ...
} >RAM
```

The key idea is that the GNU linker distinguishes between virtual (VMA) and load addresses (LMA). The VMA is the address a section has when the program is executed, the LMA is the address at which the section is loaded. For data, our linker script places the LMA of the data section within FLASH and the VMA within RAM – notice that `_sidata = _etext`.

```
ENTRY(Reset_Handler)
MEMORY
{
  RAM (rwx)  : ORIGIN = 0x20000000, LENGTH = 8K
  FLASH (rx) : ORIGIN = 0x08000000, LENGTH = 128K
}

SECTIONS
{
    .text :
    {
        KEEP(*(.isr_vector)) /* vector table */
        *(.text)             /* code */
        *(.text.*)           /* remaining code */
        *(.rodata)           /* read-only data (constants) */
        ...
    } >FLASH
   ...
   _etext = .;
   _sidata = _etext;
    /* Init data goes in RAM, but is copied after code as well */
    .data  : AT ( _sidata )
    {
       ...
        _sdata = .;
        *(.data)
       ...
        _edata = . ; /* used to init data section */
    } >RAM

.bss :
    {
       ...
        _sbss = .;    /* used to init bss */
        __bss_start__ = _sbss;
        *(.bss)
       ...
        . = ALIGN(4);
        _ebss = . ;   /* used to init bss */
        __bss_end__ = _ebss;
    } >RAM
```

Figure 3.1: Linker Script Fragment

The first portion of the `.text` section is loaded with the exception vectors (`.isr_vector`) which are later defined in the startup code. These excep-

tion vectors start at 0x08000000, as is required when the STM32 boots from flash.

There are a number of details elided that ensure the creation of labels assumed by the compiler and standard libraries as well as handling debugging sections of the binary.

The linker script and the startup code collaborate to create a meaningful executable environment. The linker script is responsible for ensuring that the various portions of the executable (e.g. the exception vectors) are in their proper place and for associating meaningful labels with specific regions of memory used by the start up code. At reset, the reset handler is called. The reset handler (defined in `startup_stm32f10x.c`) copies the initial values for variables from flash (where the linker places them) to SRAM and zeros the so-called uninitialized data portion of SRAM. (see Listing 3.2). These steps are necessary whenever the processor resets in order to initialize the "C" environment. The reset handler then calls `SystemInit` (defined in `system_stm32f10x.c` from the firmware library) which initializes the clocking system, disables and clears interrupts. The compile flag `STM32F10X_MD_VL` defined in our makefile is crucial to this code because the clock initialization code is processor specific. Finally, the reset handler calls the `main` function defined in user code. The external variables required by the reset handler to initialize memory (e.g. `_sidata, _sdata`...) are defined by the linker script.

Another important function of the startup code is to define the default interrupt vector table (Listing 3.3). In order to allow application code to conveniently redefine the various interrupt handler, every required interrupt vector is assigned an overideable (weak) alias to a default hander (which loops forever). To create a custom interrupt handler in application code, it is sufficient to define a procedure with the appropriate handler name. One word of caution – you must be careful to use exactly then names defined in the vector table for your handler or else it will not be linked into the loaded vector table !

```
// Linker supplied pointers

extern unsigned long _sidata;
extern unsigned long _sdata;
extern unsigned long _edata;
extern unsigned long _sbss;
extern unsigned long _ebss;

extern int main(void);

void Reset_Handler(void) {

   unsigned long *src, *dst;

   src = &_sidata;
   dst = &_sdata;

   // Copy data initializers

    while (dst < &_edata)
      *(dst++) = *(src++);

   // Zero bss

   dst = &_sbss;
   while (dst < &_ebss)
       *(dst++) = 0;

  SystemInit();
  __libc_init_array();
  main();
  while(1) {}
}
```

Listing 3.2: Reset Handler in `startup_stm32f10x.c`

```
static void default_handler (void) { while(1); }

void NMI_Handler (void) __attribute__ ((weak,  alias
    ↪(``default_handler'')));
void HardFault_Handler (void) __attribute__ ((weak,  alias
    ↪(``default_handler'')));
void MemMange_Handler (void) __attribute__ ((weak,  alias
    ↪(``default_handler'')));
void BusFault_Handler (void) __attribute__ ((weak,  alias
    ↪(``default_handler'')));

...

__attribute__ ((section(``.isr_vector'')))

void (* const g_pfnVectors[])(void) = {
        _estack,
        Reset_Handler,
        NMI_Handler,
        HardFault_Handler,

...
```

Listing 3.3: Interrupt Vectors

# Chapter 4

# STM32 Configuration

The STM32 processors are complex systems with many peripherals. Before any of these peripherals can be used they must be configured. Some of this configuration is generic – for example clock distribution and pin configuration – while the rest is peripheral specific. Throughout this chapter, we utilize a simple "blinking lights" program as a guiding example.

The fundamental initialization steps required to utilize any of the STM32 peripherals are:

1. Enable clocks to the peripheral
2. Configure pins required by the peripheral
3. Configure peripheral hardware

The STM32 processors, as members of the Cortex-M3 family, all have a core system timer which can be used to provide a regular timing "tick." We utilize this timer to provide a constant blink rate for our example. The overall structure of this program is illustrated in Figure 4. The program begins by including the relevant firmware library headers – in this case for clock and pin configuration. The main routine follows the initialization steps described above and then enters a loop in which it toggles an LED and waits for 250ms. Procedure main is followed by code implementing the delay function which utilizes the system timer. Finally, a helper function is provided to handle assertion violations in the firmware library (required if `USE_FULL_ASSERT` is defined when compiling firmware library modules). While the assert_failed handler does nothing, it is very useful when debugging new projects as the

firmware library will perform extensive parameter checking. In the event of an assertion violation, GDB can be used to examine the parameters of this routine to determine the point of failure.

```
#include <stm32f10x.h>
#include <stm32f10x_rcc.h>
#include <stm32f10x_gpio.h>

void Delay(uint32_t nTime);

int main(void)
{
  GPIO_InitTypeDef  GPIO_InitStructure;

  // Enable Peripheral Clocks
  ... (1) ...
  // Configure Pins
  ... (2) ...
  // Configure SysTick Timer
  ... (3) ...
  while (1) {
     static int ledval = 0;

     // toggle led
    ... (4) ...

     Delay(250);  // wait 250ms
  }
}

// Timer code
... (5) ...

#ifdef  USE_FULL_ASSERT
void assert_failed(uint8_t* file, uint32_t line)
{
  /* Infinite loop */
  /* Use GDB to find out why we're here */
  while (1);
}
#endif
```

Figure 4.1: Blinking Lights

The STM32 VL discovery board has an LED driven by I/O pin PC9. [14] In order to configure this pin, clocks must first be enabled for GPIO

Port C with the following library command (described in greater detail in Section 4.1).

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE); // (1)
```

After enabling the clocks, it is necessary to configure any required pins. In this case, a single pin (PC9) must be configured as an output (described in greater detail in Section 4.2).

```
/* (2) */

GPIO_StructInit(&GPIO_InitStructure);
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
GPIO_Init(GPIOC, &GPIO_InitStructure);
```

Once configuration is complete, the pin can be set or reset with the following code:

```
/* (4) */
GPIO_WriteBit(GPIOC, GPIO_Pin_9, (ledval) ?  Bit_SET : Bit_RESET);
ledval = 1 - ledval;
```

The blinking light demo also utilizes a "timer tick" to measure the passage of time. While this timer tick utilizes interrupts, which we will not be discussing until Chapter 11, the actual use here can be treated simply as an idiom. The Cortex-M3 core used in the STM32 processors has a dedicated timer for this function. The timer as a multiple of the system clock (which is defined in ticks/second) – here we configure it for 1 msec interrupts (the constant `SystemCoreClock` is defined in the firmware library to be the number of system clock cycles per second):

```
/* (3) */
if (SysTick_Config(SystemCoreClock / 1000))
   while (1);
```

Every 1 msec, the timer triggers a call to the `SysTick_Handler`. For the blinking light demo, we simply decrement a shared counter – declared as `__IO` to ensure that the compiler doesn't perform undesired optimization.

```
  /* (5) */
static __IO uint32_t TimingDelay;

void Delay(uint32_t nTime){
  TimingDelay = nTime;
  while(TimingDelay != 0);
}

void SysTick_Handler(void){
  if (TimingDelay != 0x00)
    TimingDelay--;
}
```

This simple blinking lights program requires support from two library modules (`stm32_gpio.c`, `stm32_rcc.c`). To include these in the project, we have to slightly modify the Makefile provided with the demo project.

```
TEMPLATEROOT = ../../stm32vl_template

# additional compilation flags

CFLAGS  += -O0 -g
ASFLAGS += -g

# project files

OBJS=    $(STARTUP) main.o
OBJS+=   stm32f10x_gpio.o stm32f10x_rcc.o

# include common make file

include $(TEMPLATEROOT)/Makefile.common
```

In the remainder of this chapter we examine clock and pin configuration in greater detail.

### Exercise 4.1 Blinking Lights

Complete the Blinking lights main.c and create a project using the demo program described in Chapter 3 as an example. You should compile and run your program.

Modify your program to cause an assertion violation – for example replacing GPIOC with 66 when initializing the pin – and use GDB to find the place in the library source code where an assertion failed.

## 4.1 Clock Distribution

In the world of embedded processors, power consumption is critical; hence, most sophisticated embedded processors provide mechanisms to power down any resources that are not required for a particular application. The STM32 has a complex clock distribution network which ensures that only those peripherals that are actually needed are powered. This system, called Reset and Clock Control (RCC) is supported by the firmware module `stm32f10x_rcc.[ch]`. While this module can be used to control the main system clocks and PLLs, any required configuration of those is handled by the startup code provided with the examples in this book. Our concern here is simply with enabling the peripheral clocks.

The STM32 peripherals are organized into three distinct groups called APB1, APB2, and AHB. APB1 peripherals include the I2C devices, USARTs 2-5, and SPI devices; APB2 devices include the GPIO ports, ADC controllers and USART 1. AHB devices are primarily memory oriented including the DMA controllers and external memory interfaces (for some devices)

Clocks to the various peripherals can be controlled with three firmware routines:

```
RCC_APB1PeriphClockCmd(uint32_t RCC_APB1PERIPH,
                FunctionalState NewState)
RCC_APB2PeriphClockCmd(uint32_t RCC_APB2PERIPH,
                FunctionalState NewState)
RCC_AHBPeriphClockCmd(uint32_t RCC_AHBPERIPH,
                FunctionalState NewState)
```

Each routine takes two parameters – a bit-vector of peripherals whose state should be modified, and an action – ENABLE or DISABLE. For example, GPIO ports A and B can be enabled with the following call:

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA |
                       RCC_APB2Periph_GPIOB, ENABLE);
```

The appropriate constants are defined in `stm32f10x_rcc.h`; the constant names, shown in Table 4.1, are relatively self-explanatory and correspond to the device names utilized in the various STM32 reference manuals ([20, 21]), limited to those that are present in the STM32 MCU on the discovery board. It is a testament to the STM32 family design that the same constants and core library applies across a broad family of devices. (Note the stm32f2xx and stm32f4xx components have different, although similar standard peripheral libraries)

| APB1 Devices | APB2 Devices |
|---|---|
| RCC_APB1Periph_BKP | RCC_APB2Periph_ADC1 |
| RCC_APB1Periph_CEC | RCC_APB2Periph_AFIO |
| RCC_APB1Periph_DAC | RCC_APB2Periph_GPIOA |
| RCC_APB1Periph_I2C1 | RCC_APB2Periph_GPIOB |
| RCC_APB1Periph_I2C2 | RCC_APB2Periph_GPIOC |
| RCC_APB1Periph_PWR | RCC_APB2Periph_GPIOD |
| RCC_APB1Periph_SPI2 | RCC_APB2Periph_GPIOE |
| RCC_APB1Periph_TIM2 | RCC_APB2Periph_SPI1 |
| RCC_APB1Periph_TIM3 | RCC_APB2Periph_TIM1 |
| RCC_APB1Periph_TIM4 | RCC_APB2Periph_TIM15 |
| RCC_APB1Periph_TIM5 | RCC_APB2Periph_TIM16 |
| RCC_APB1Periph_TIM6 | RCC_APB2Periph_TIM17 |
| RCC_APB1Periph_TIM7 | RCC_APB2Periph_USART1 |
| RCC_APB1Periph_USART2 | |
| RCC_APB1Periph_USART3 | |
| RCC_APB1Periph_WWDG | |
| AHB Devices | |
| RCC_AHBPeriph_CRC | RCC_AHBPeriph_DMA |

Table 4.1: Clock Distribution Constant Names (`stm32f10x_rcc.h`)

The standard peripheral library code to enable clocks doesn't perform any "magic", but rather releases the programmer from the need to be intimately familiar with the micro-controller registers. For example, the APB2 peripherals are enabled through a single register (RCC_APB2ENR) (illustrated in Figure 4.2) [1].); each peripheral is enabled (or disabled) by the state of a single bit. For example, Bit 2 determines whether GPIOA is enabled (1) or disabled (0). Applying the structure and register definitions in `<stm32f10x.h>` we can enable GPIOA and GPIOB as follows:

```
ABP2ENR |= 0x0C;
```

[1] The various control registers are fully documented in [20]

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Reserved | | | | | | | | | | | | | TIM17 | TIM16 | TIM15 |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Res. | URT1 | Res. | SPI1 | TIM1 | Res. | ADC1 | IOPG | IOPF | IOPE | IOPD | IOPC | IOPB | IOPA | Res. | AFIO |

Figure 4.2: APB2 Peripheral Clock Enable Register

## 4.2 I/O Pins

Most of the pins of the STM32 can be configured as input or output and may be connected to either the GPIO ports or "alternative functions" (other peripherals). As a standard naming convention, the pins are called by their GPIO function – for example PA0 (bit 0 of port A) or PB9 (bit 9 of port B). Indeed, the labeling of the discovery board follows this convention. Subject to specific hardware constraints, each pin can be configured in the modes illustrated in Figure 4.2.

| Function | Library Constant |
|---|---|
| Alternate function open-drain | `GPIO_Mode_AF_OD` |
| Alternate function push-pull | `GPIO_Mode_AF_PP` |
| Analog | `GPIO_Mode_AIN` |
| Input floating | `GPIO_Mode_IN_FLOATING` |
| Input pull-down | `GPIO_Mode_IPD` |
| Input pull-up | `GPIO_Mode_IPU` |
| Output open-drain | `GPIO_Mode_Out_OD` |
| Output push-pull | `GPIO_Mode_Out_PP` |

Table 4.2: Pin Modes (`stm32f10x_gpio.h`)

By default, most pins are reset to "Input Floating" – this ensures that no hardware conflicts will occur when the system is powering up. The firmware library provides an initialization routine in `stm32f10x_gpio.[ch]` which may be used to reconfigure the pins. For example, for the blinking lights we configured PC9 as a (slow) output as illustrated in Listing 4.1.

When configuring an output as shown above, we have three choices of output "speed" – 50 MHz, 10 MHz, and 2 MHz. In general, for reasons of power consumption and noise, it is desirable to use the lowest speed consistent

```
// see stm32f10x_gpio.h
GPIO_InitTypeDef  GPIO_InitStructure;

GPIO_StructInit(&GPIO_InitStructure);

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
GPIO_Init(GPIOC, &GPIO_InitStructure);
```

Listing 4.1: PC9 Configuration

with the I/O requirements. The field `GPIO_Pin` is a bit vector and it is possible to configure multiple pins associated with a single port in a single step.

**Note:** The pin-outs for specific parts (including the assignment of peripherals to pins) are defined in their respective data sheets (e.g. [15]) and NOT in the programming manual ! The pin assignments for the Discovery board are documented in the user's manual [14].

As we have seen, we can write a value to the pin controlling the LED with the following procedures:

```
GPIO_WriteBit(GPIOC, GPIO_Pin_9, x); // x is Bit_SET or Bit_RESET
```

The gpio library module provides procedures to read and write both individual pins and entire ports – the later is especially helpful when capturing parallel data. It is instructive to read `stm32f10x_gpio.h`.

To (re-)configure the pin associated with the discovery board push button, we can use the following code (after configuring the clock for Port A !):

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
GPIO_Init(GPIOA, &GPIO_InitStructure);
```

To read the current value of the pushbutton we can execute:

```
GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_0)
```

Recall from Chapter 2 that each GPIO port is controlled by 7 registers:

```
typedef struct
{
```

```
  volatile uint32_t CRL;
  volatile uint32_t CRH;
  volatile uint32_t IDR;
  volatile uint32_t ODR;
  volatile uint32_t BSRR;
  volatile uint32_t BRR;
  volatile uint32_t LCKR;
} GPIO_TypeDef;
```

The 16 bits of each port are configured with `CRL` (bits 0-7) and `CHR` (pins 8-15). To support the various I/O modes, 4 configuration bits are required for each GPIO bit. The 16 GPIO bits can be read in parallel (`IDR`) and written in parallel (`ODR`). As a convenience, registers `BSRR` and `BRR` provide a mechanism to set and reset individual bits. The lock register `LCKR` provides a mechanism to "lock" the configuration of individual bits against software reconfiguration and hence protect hardware from software bugs.

**Exercise 4.2 Blinking Lights with Pushbutton**

Modify the blinking lights program to additionally track the state of the user pushbutton (PA0) on the blue LED (PC8). See if you can figure out how to configure both LEDs with a single call to `GPIO_Init`.

## 4.3 Alternative Functions

Peripherals such as the USARTs share pins with the GPIO devices. Before these peripherals can be utilized, any outputs required by the peripheral must be configured to an "alternative mode". For example, the Tx pin (data out) for USART1 is configured as follows:

```
GPIO_InitStruct.GPIO_PIN = GPIO_Pin_9;
GPIO_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStruct.GPIO_Mode = GPIO_Mode_AF_PP;
GPIO_Init(GPIOA, &GPIO_InitStruct);
```

The specific configuration required for each peripheral is described in section 7.1.11 of the stm32f10xx reference manual RM0041 [20] (section 9.1.11 for stm32f103xx reference manual RM0008 [21]).

## 4.4 Remapping

It is also possible to "remap" pins so that non-default pins are used for various peripherals in order to minimize conflicts. These re-mappings, which

are beyond the scope of this book, are described in the appropriate STM32 reference manual ([20, 21])

## 4.5 Pin Assignments For Examples and Exercises

In this book we develop a series of examples and exercises based upon the STM32VL Discovery board. Ensuring that these examples can work together required some care in the selection of STM32 devices and GPIO pins – for example we used the SPI2 device rather than the SPI1 device due to a resource conflict. In Table 4.5 we enumerate all the pin assignments used in this book along with the configurations required for these uses.

| Device | Pin | Function | Configuration |
|---|---|---|---|
| User Button | PA0 | | Input floating |
| LCD Backlight | PA1 | Backlight | Output/Alternative function push-pull |
| DAC1 | PA4 | DAC Output | Analog |
| ADC | PA6 | IN6 | Input floating |
| | PA7 | IN7 | Input floating |
| Timer 1 | PA8 | Channel 1 | Input floating |
| USART 1 | PA9 | TX | Alternative function push-pull |
| | PA10 | RX | Input Pull-up |
| | PA11 | nCTS | Input Pull-up |
| | PA12 | nRTS | Alternative function push-pull |
| Timer 3 | PB0 | Channel 3 | Alternative function push-pull |
| | PB1 | Channel 4 | Alternative function push-pull |
| I2C1 | PB6 | SCK | Alternative function open-drain |
| | PB7 | SDA | Alternative function open-drain |
| Timer 4 | PB9 | Channel 4 | Alternative function push-pull |
| I2C2 | PB10 | SCK | Alternative function open-drain |
| | PB11 | SDA | Alternative function open-drain |
| SPI2 | PB13 | CLK | Alternative function push-pull |
| | PB14 | MISO | Input pull-up |
| | PB15 | MOSI | Alternative function push-pull |
| LCD control | PC0 | LCD Select | Output push-pull |
| | PC1 | Reset | Output push-pull |
| | PC2 | Data/Control | Output push-pull |
| SD Card | PC6 | Select | Output push-pull |
| Blue LED | PC8 | | Output push-pull |
| Green LED | PC9 | | Output push-pull |
| SPI EEPROM CS | PC10 | | Output push-pull |

Table 4.3: Pin Assignments for Exercises

## 4.6 Peripheral Configuration

As mentioned the third configuration stage, after clock distribution and pin configuration, is peripheral configuration. While we defer the discussion of peripheral specific configuration, the standard firmware library offers a standard pattern for the configuration process. We've see a bit of this already with GPIO configuration where a device specific structure was populated with a set of parameters and one or more pins for a given port were initialized:

```
GPIO_StructInit(&GPIO_InitStructure);
... fill in structure ...
GPIO_Init(GPIOx, &GPIO_InitStructure);
```

It is also possible to "de-initialize" a port, returning all of its pin configuration to the hardware reset state with a call to

```
GPIO_DeInit(GPIOx)
```

The DeInit function resets the peripheral registers, but it does not disable the peripheral clock – that requires a separate call to the clock command (with DISABLE replacing ENABLE).

This pattern – an initialization structure, an init function, and a de-init function is repeated throughout the standard peripheral library. The basic naming convention for peripheral "ppp" is:

| | |
|---|---|
| Files | `stm32f10x_ppp.[c\|h]` |
| Init Structure | `ppp_InitTypeDef` |
| Zero Structure | `ppp_StructInit(ppp_InitTypeDef*)` |
| Initialize Peripheral | `ppp_Init([sub-device,] ppp_InitTypeDef*)` |
| De-initialize Peripheral | `ppp_DeInit([sub-device])` |

Examples of devices with the optional "sub device" are USART, SPI, I2C. Timers are a somewhat complicated case because each timer is typically multiple devices – a "time base", zero or more output compares, zero or more input captures. There are other exceptions, but mostly for peripherals that are not supported on the medium density value-line parts.

### Exercise 4.3 Configuration without Standard Peripheral Library

Write a program using only the constants defined in the programmer reference manual ([20]) that: configures the pins for the user push-button and blue LED, and, in an infinite loop, displays the button state on the LED.

Your code should look like

```
main()
{
   // configure button
   // configure led

   while (1)
   {
     if (read(button))
       led = on;
     else
       led = off;
   }
}
```

# Chapter 5

# Asynchronous Serial Communication

After LEDs and pushbuttons, the most basic method for communication with an embedded processor is asynchronous serial. Asynchronous serial communication in its most primitive form is implemented over a symmetric pair of wires connecting two devices – here I'll refer to them as the host and target, although those terms are arbitrary. Whenever the host has data to send to the target, it does so by sending an encoded bit stream over its transmit (TX) wire; this data is received by the target over its receive (RX) wire. Similarly, when the target has data to send to the host it transmits the encoded bit stream over its TX wire and this data is received by the host over its RX wire. This arrangement is illustrated in Figure 5. This mode of communications is called "asynchronous" because the host and target share no time reference. Instead, temporal properties are encoded in the bit stream by the transmitter and must be decoded by the receiver.

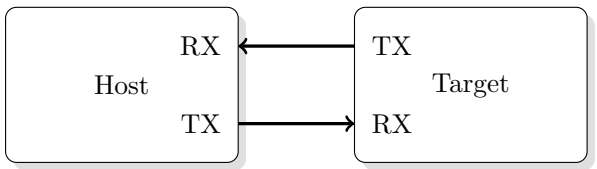



Figure 5.1: Basic Serial Communications Topology

A commonly used device for encoding and decoding such asynchronous bit streams is a Universal Asynchronous Receiver/Transmitter (UART) which converts data bytes provided by software into a sequence of individual bits and, conversely, converts such a sequence of bits into data bytes to be passed off to software. The STM32 processors include (up to) five such devices called USARTs (for universal synchronous/asynchronous receiver/transmitter) because they support additional communication modes beyond basic asynchronous communications. In this Chapter we will explore serial communication between the (target) STM32 USART and a USB/UART bridge connected to a host PC.

UARTs can also be used to interface to a wide variety of other peripherals. For example, widely available GSM/GPRS cell phone modems and Bluetooth modems can be interfaced to a micro-controller UART. Similarly GPS receivers frequently support UART interfaces. As with the other interfaces we'll consider in future chapters, the serial protocol provides access to a wide variety of devices.

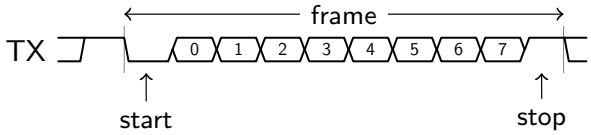



Figure 5.2: Serial Communications Protocol

One of the basic encodings used for asynchronous serial communications is illustrated in Figure 5. Every character is transmitted in a "frame" which begins with a (low) start bit followed by eight data bits and ends with a (high) stop bit. The data bits are encoded as high or low signals for (1) and (0), respectively. Between frames, an idle condition is signaled by transmitting a continuous high signal. Thus, every frame is guaranteed to begin with a high-low transition and to contain at least one low-high transition. Alternatives to this basic frame structure include different numbers of data bits (e.g. 9), a parity bit following the last data bit to enable error detection, and longer stop conditions.

There is no clock directly encoded in the signal (in contrast with signaling protocols such as Manchester encoding) – the start transition provides the only temporal information in the data stream. The transmitter and receiver

each independently maintain clocks running at (a multiple of) an agreed frequency – commonly, and inaccurately, called the baud rate. These two clocks are not synchronized and are not guaranteed to be exactly the same frequency, but they must be close enough in frequency (better than 2%) to recover the data.

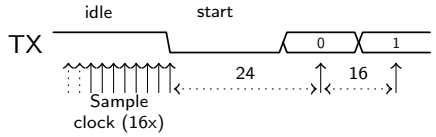



Figure 5.3: UART Signal Decoding

To understand how the receiver extracts encoded data, assume it has a clock running at a multiple of the baud rate (e.g. 16x) starting from an idle state as illustrated in Figure 5. The receiver "samples" its RX signal until it detects a high-low transition. It then waits 1.5 bit periods (24 clock periods) to sample its RX signal at what it estimates to be the center of data bit 0. The receiver then samples RX at bit-period intervals (16 clock periods) until it has read the remaining 7 data bits and the stop bit. From that point the process repeats. Successful extraction of the data from a frame requires that, over 10.5 bit periods, the drift of the receiver clock relative to the transmitter clock be less that 0.5 periods in order to correctly detect the stop bit.

### Exercise 5.1 Testing the USB/UART Interface

Before we discuss the implementation of UART communications with the STM32, it may be helpful to use the Saleae Logic to "see" the asynchronous signals. We will use the USB-UART bridge to generate signals that we will then capture with logic analyzer. Later, this will be an essential tool for debugging your code executing on the STM32.

It is extremely difficult to debug hardware device drivers using only a software debugger such as GDB because these debuggers provide no visibility into what the hardware is actually doing. For example, if one neglects to configure the clocks to a specific hardware peripheral, that peripheral will do nothing in response to software commands even if all the peripheral specific software is completely correct. Perhaps even more vexing is when the hardware interface almost works. For example, in developing the I2C examples

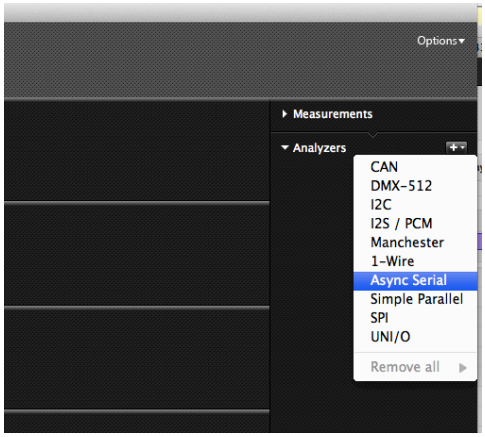



Figure 5.4: Adding A Protocol Analyzer

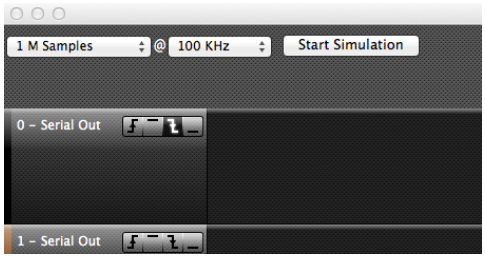



Figure 5.5: Configuring Capture Rate and Trigger

using the Wii Nunchuk, I found that while the STM32 and Nunchuk were communicating, the data exchanged was incorrect – ultimately, I tracked this to an incorrect clock rate. In this case GDB was slightly helpful it isolating but not diagnosing the problem.

For developing embedded systems with communicating hardware, a logic analyzer provides the capability of observing the actual communication events between two hardware components. Essentially, a logic analyzer "listens" to a set of digital signals until a trigger event occurs and then captures a window of data around the trigger event. The software associated with a logic analyzer allows us to interpret the captured data. In the simplest case a trigger event might be a the occurrence of a transition on a signal – in the case of USART communication, every transmission event begins with a falling transition. Throughout this book we use the Saleae Logic analyzer. Professional

logic analyzers can watch for complex sequences of events to form a trigger condition, but the Saleae analyzer is limited to simple events. Furthermore, the Saleae can only capture signals at a modest sample rate. However, it is more than sufficient for the examples we present.

Early logic analyzers were relatively limited in the display of captured information as a simple time sequence of signals. More modern systems, including the Saleae Logic, provide protocol analysis capabilities which overlay the time sequences with interpretations. For example, with serial protocol analyzers, the actual characters transmitted are displayed superimposed over the time sequence.

The Saleae Logic software provides a simulation mode which is useful for learning how the system works. We will begin by using the simulation mode to evaluate serial protocol. You will need to download and install the Saleae logic software appropriate for your system `http://www.saleae.com/downloads/`.

Launch the Logic software. On the right side of the screen "add" an "Async Serial" analyzer – the default options are fine – and rename the channel 0 to "Serial Out." This is illustrated in Figure 5.4.

Set the capture rate to 100 Khz, the number of samples to 1 M, and the trigger condition on Serial Out to the falling arrow (a transition from '1' to '0') as illustrated in Figure 5.5. When configuring Saleae for other protocols, you will need to select a capture rate that is a multiple (ideally 10 or more) of the data rate. You will find that the Saleae logic is somewhat limiting in this regard. For example, I have found it necessary to debug the SPI protocol at a somewhat reduced rate in order to ensure that the Saleae logic is capable of buffering the data. Ultimately, it may be necessary to debug at a modest rate and, when confident everything is working, increase the underlying protocol speed.

Finally, start the simulation. After the simulation is complete zoom in on the data (by clicking with the mouse pointer) until individual serial frames are visible. You may wish to read the Saleae Logic User's Guide to learn how to navigate the displayed data.

Once you understand the basic interface, connect the ground and TX pins of a USB/UART adaptor to the gray and black (channel 0) wires of the logic. Connect the USB/UART adaptor and Saleae Logic module to your computer. We will use the Unix "screen" program as an interface to the USB-UART bridge.

First you need to determine the device name for the UART-USB bridge. List the contents of /dev/tty* to find all the tty devices. You are looking for a

device with USB in its name (e.g. in OS X, `tty.SLAB_USBtoUART`, or `ttyUSB0` in Linux).

Once you've found the device (e.g. /dev/ttyXXX), it's important to make sure that it is correctly configured. To determine the current configuration execute

```
stty -f /dev/ttyXXX
```

On Linux `-f` should be replaced by `-F`. This will list the current configuration. If the baud rate is other than 9600, you have two choices – change the baud rate in your program, or modify the device baud rate (see the man page for `stty` to learn how to do this.). Once the configuration of your program and device match, execute the following in the screen terminal.

```
screen /dev/ttyXXX
```

Now, anything you type in the screen program will be transmitted by the USB/UART adaptor. In the logic program, "start" a capture – you should see a window that says "waiting for a trigger". Then, in the screen window type the alphabet as quickly as you can. Once the capture is complete, zoom in and examine the captured data – you should see the characters you typed. If the data is marked with framing errors, then it is likely that the USB/UART is transmitting at a baud rate that is different than the Saleae Logic expects. Correct this (either by reconfiguring Logic, or the USB/UART) before proceeding.

To exit from screen type `C-a k\` ("control a k").

## 5.1 STM32 Polling Implementation

The simplest form of USART communication is based upon polling the state of the USART device. Each STM32 USART has 6 registers – 4 of which are used for configuration through the initialization routines provided by the driver library as shown in Section 5.2. The remaining 2 registers are the "data register" and the "status register". While the data register occupies a single memory word, it is really two separate locations; when the data register is written, the written character is transmitted by the USART. When the data register is read, the character most recently received by the USART is returned. The status register contains a number of flags to determine the current USART state. The most important of these are:

```
USART_FLAG_TXE  -- Transmit data register empty
USART_FLAG_RXNE -- Receive data register not empty
```

Other flags provide error information include parity, framing, and overrun errors which should be checked in a robust implementation.

In order to transmit a character, the application software must wait for the transmit data register to be empty and then write the character to be transmitted to that register. The state of the transmit data register is determined by checking the `USART_FLAG_TXE` flag of the USART status register. The following code implements a basic `putchar` procedure utilizing USART1.

```
int putchar(int c){
 while (USART_GetFlagStatus(USART1, USART_FLAG_TXE) == RESET);
 USART1->DR =  (c & 0xff);
 return 0;
}
```

With a single transmit data register, this implementation is only as fast as the underlying baud rate and must wait (by "polling" the flag status) between characters. For simple programs this may be acceptable, but in general a non-block implementation such as the one we will describe in Section 11.5 is preferred. The dual of `putchar` is `getchar`.

```
int getchar(void){
 while (USART_GetFlagStatus(USART1, USART_FLAG_RXNE) == RESET);
 return  USART1->DR & 0xff;
}
```

Notice the use of the status flag `USART_IT_RXNE` to determine when there is a character to receive ("receive data not empty"). Also, notice the use of a mask to select the lower 8 bits returned by `USART_ReceiveData` – this library procedure returns 9 bits by default because configuration of the USART permits 9-bit data. For example, the 9th bit might contain parity information which could be used to check the validity of the received character.

While the polling implementation of `putchar` has the deficiency of being slow, the polling implementation of `getchar` has a fatal flaw – if the application code does not receive characters as they arrive, but the host continues to send characters, an "overrun" will occur that results in lost characters. The STM32 is has only a single receive data buffer while many micro-controllers have 8 or 16 character buffers. This provides very little room for timing variation in the application code that is responsible for monitoring the USART receiver.

In the Chapter 11 we discuss the use of interrupt driven code to alleviate this deficiency.

## 5.2 Initialization

As with all STM32 peripherals, the USARTs must be initialized before they can be used. This initialization includes pin configuration, clock distribution, and device initialization. Initialization is handled most conveniently with the Standard Peripheral Driver library – in the following we assume version 3.5.0. The stm32f100 component in the discovery board has 3 USARTs called USART1 – USART3 throughout the documentation and driver library. In the following we will be utilizing USART1, but the principles for the other USARTs are the same.

There are three modules (in addition to the general header) that are part of the driver library which are required for USART applications (you will need to include the associated object files in your make file).

```
#include <stm32f10x.h>
#include <stm32f10x_gpio.h>
#include <stm32f10x_rcc.h>
#include <stm32f10x_usart.h>
```

The first initialization step is to enable the RCC (reset and clock control) signals to the various functional blocks required for utilizing the USART – these include GPIO ports (port A for USART1), the USART component and the AF (alternative function) module. For USART1, the necessary RCC configuration step is:

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1 |
                       RCC_APB2Periph_AFIO |
                       RCC_APB2Periph_GPIOA, ENABLE);
```

USART2 and USART3 are "APB1" peripherals, hence their RCC initialization differs slightly. Notice the that various APB2 flags are or'd together; it is also acceptable to enable the clocks in separate steps.

Once clocks are enabled, it is necessary to configure the USART pins – the default pins for all three USARTs are provided in Table 5.1.

The STM32 reference manuals provide key information for configuring the GPIO pins for the various devices. For the USARTs, this information is reproduced here as Table 5.2. More complete pin configuration information is available from the device data sheet [15].

| Function | Pin x=1 | x=2 | x=3 |
|---|---|---|---|
| USARTx_TX | PA9 | PA2 | PB10 |
| USARTx_RX | PA10 | PA3 | PB11 |
| USARTx_CK | PA8 | PA4 | PB12 |
| USARTx_RTS | PA12 | PA1 | PB14 |
| USARTx_CTS | PA11 | PA0 | PB13 |

Table 5.1: USART Pins

| USART pinout | Configuration | GPIO Configuration |
|---|---|---|
| USARTx_TX | Full Duplex | Alternate function push-pull |
| | Half duplex Synchronous mode | Alternate function push-pull |
| USARTx_RX | Full Duplex | Input floating/Input Pull-up |
| | Half duplex Synchronous mode | Not used. Can be used as General IO |
| USARTx_CK | Synchronous mode | Alternate function push-pull |
| USARTx_RTS | Hardware flow control | Alternate function push-pull |
| USARTx_CTS | Hardware flow control | Input floating/Input pull-up |

Table 5.2: USART Pin Configuration

As mentioned previously, the USARTs in the STM32 are capable of supporting an additional operational mode – synchronous serial – that requires an separate clock signal (`USARTx_CK`) which we will not be using. In addition, the USARTs have the capability to support "hardware flow control" (signals `USARTx_RTS` and `USARTx_CTS`) which we will discuss in Section 11.5. For basic serial communications we must configure two pins – USART1_Tx and USART1_Rx. The former is an output "driven" by the USART component (an "alternative function" in the STM32 documentation) while the later is an input which may be configured as floating or pulled up. Pin configuration is performed with functions and constants defined in `stm32f10x_gpio.h`.

```
GPIO_InitTypeDef GPIO_InitStruct;

GPIO_StructInit(&GPIO_InitStruct);

 // Initialize USART1_Tx

GPIO_InitStruct.GPIO_Pin = GPIO_Pin_9;
GPIO_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStruct.GPIO_Mode = GPIO_Mode_AF_PP;
GPIO_Init(GPIOA, &GPIO_InitStruct);

 // Initialize USART1_RX

GPIO_InitStruct.GPIO_PIN = GPIO_Pin_10;
GPIO_InitStruct.GPIO_Mode = GPIO_Mode_IN_FLOATING;
GPIO_Init(GPIOA, &GPIO_InitStruct);
```

The final initialization step is to configure the USART. We use the default USART initialization of 9600 "baud", 8 data bits, 1 stop bit, no parity, and no flow control provided by the library procedure `USART_StructInit`. Changes to the default initialization are made by modifying specific fields of the `USART_InitStructure`. [1]

```
// see stm32f10x_usart.h

USART_InitTypeDef USART_InitStructure;

// Initialize USART structure

USART_StructInit(&USART_InitStructure);

// Modify USART_InitStructure for non-default values, e.g.
//  USART_InitStructure.USART_BaudRate = 38400;

USART_InitStructure.USART_BaudRate = 9600;
USART_InitStructure.USART_Mode  = USART_Mode_Rx | USART_Mode_Tx;
USART_Init(USART1,&USART_InitStructure);
USART_Cmd(USART1, ENABLE);
```

### Exercise 5.2 Hello World!

The code in the preceding sections provide all of the basic functionality required to successfully use an STM32 USART. In this section, I will guide you

[1] As with many of the library procedures, it can be illuminating to read the source code – in this case the module `stm32f10x_usart.c`.

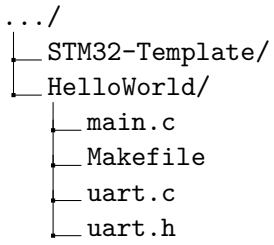



Figure 5.6: Hello World Project

```
int uart_open(USART_TypeDef* USARTx, uint32_t baud, uint32_t flags);
int uart_close(USART_TypeDef* USARTx);
int uart_putc(int c, USART_TypeDef* USARTx);
int uart_getc(USART_TypeDef* USARTx);
```

Listing 5.1: Uart Interface

through the process of developing a simple application that repeatedly sends the string "hello world" from the STM32 to a host computer. This is the first application that requires actually wiring hardware components together, so I will walk you through that process. In the following I assume you have access to a USB/UART bridge as described in Section 1.1.

You will be developing a program using the build environment provided with the blinking light example. The directory structure for the (complete) project is illustrated in Figure 5.6

The `uart.[ch]` files provide the basic software interface to the stm32 USART1. Your first task is to implement the functions specified by the `uart.h` file (Listing 5.1).

The function `uart_open` should

1. Initialize usart/gpio clocks.
2. Configure usart pins
3. Configure and enable the usart1

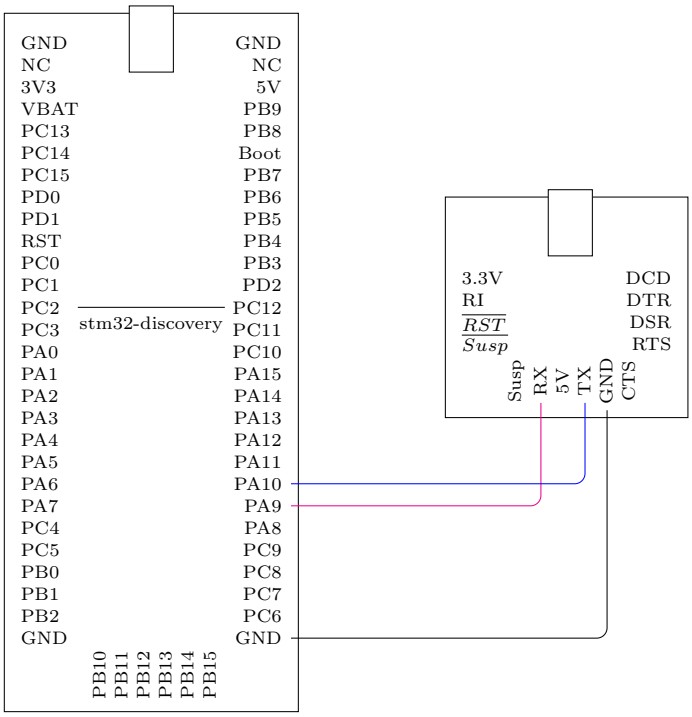



Figure 5.7: Wiring Uart Polling

You will need the following "include" directives in your `uart.c` file:

```
#include <stm32f10x.h>
#include <stm32f10x_rcc.h>
#include <stm32f10x_gpio.h>
#include <stm32f10x_usart.h>
#include "uart.h"
```

Functions `uart_putc` and `uart_getc` should read or write individual characters, respectively. They should behave like the standard Linux `getc` and `putc`. Your `main.c` file should:

1. Initialize the timer tick.
2. Initialize the uart.
3. Write `Hello World!\n\r''` every 250 msec.

You will need to include `stm32f10x.h`, `stm32f10x_usart.h`, and `uart.h` in your `main.c` file

You should copy and modify the `Makefile` from the `BlinkingLight` directory. You should add `uart.o` and `stm32f10x_usart.o` to the OBJS list of project files. You may need to modify the variable `TEMPLATEROOT` to point to the directory containing the Demo program.

At this point you should make sure your project compiles correctly (type make !). If compilation succeeded, you should have an executable named "HelloWorld.elf".

There are only three wires that need to be connected as illustrated in Figure 5.7. The ground signals of the discovery board and uart bridge need to be connected (these are labeled `gnd`) – there are multiple ground pins on the discovery board all of which are electrically connected by the PCB. Thus, only a single wire (black is traditional) is needed to tie the ground signals together. You will need to connect the rx (tx) pin of USART1 to the tx (rx) pin of the USB/uart.

To test your program, open three terminal windows. One will be used to communicate with the USB/UART adaptor, a second will be used to execute the gdb server, and the third to execute gdb.

Start the screen program to communicate with the USB/UART adaptor. In the gdbserver window, execute

```
st-util -1
```

Finally, in the gdb terminal, navigate to the directory with your program and execute

```
arm-none-eabi-gdb HelloWorld.elf
(gdb) target extended-remote :4242
(gdb) load
(gdb) continue
```

With some luck you will be rewarded with "hello world" in the screen window. If not, it's time for some serious debugging.

Even if you do not encounter problems, it will be instructive to use the Saleae Logic to observe the low-level behavior of the STM32 UART. To capture data from the lab board it is necessary to connect two wires from the Saleae Logic to your circuit – ground (gray) and channel 0 (black). The ground wire should be connected to one of the GND pins and channel 0 to the

USART1 tx (PA9) pin. Connect the Saleae logic analyzer to your computer and configure the software for Async serial on channel 0.

Compile your project and download your executable (follow the directions in Section 1.2). Start the capture process on the Saleae logic, and run your program. With luck, you will be rewarded with a capture of `Hello World\n\r` being transmitted. If no data is captured, carefully check that your software correctly configures the necessary peripherals and then, using GDB, try to determine if your code is actually attempting to transmit the string.

### Exercise 5.3 Echo

Modify your program to receive and echo input from the host. There are two ways to do this – line at a time and character at a time (try both !). Try testing your code using cat to send an entire file to your program at once while capturing the output in another file. You can check your output using "diff." It is likely that the input and output don't match (especially if you perform the echo on a line at a time basis). We will return to this problem in Chapter 11.

# Chapter 6

# SPI

The SPI bus is widely used serial interface for communicating with many common hardware devices including displays, memory cards, and sensors. The STM32 processors have multiple SPI interfaces and each of these interfaces can communicate with multiple devices. In this chapter we show how to use the SPI bus to interface to widely available EEPROM memory chips. These chips provide non-volatile storage for embedded systems and are frequently used to store key parameter and state data. In Chapter 7 we show how to use the SPI bus to interface with a color LCD display module and in Chapter 8 we will use the same interface to communicate with a commodity flash memory card.

## 6.1 Protocol

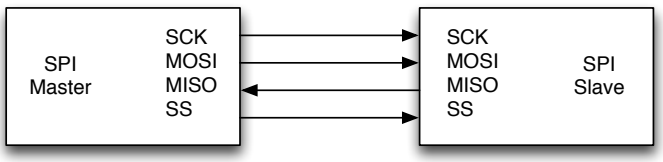



Figure 6.1: SPI Protocol Block Diagram

The basic SPI interface is illustrated in Figure 6.1. Every instance of a SPI bus has a single master and one or more slaves. While the STM32 can be configured in either role, here we consider only the case where it is configured as a master. In the figure there are four single-wire signals – three from the

master and one from the slave. One of these signals, `SS` (for "slave select") must be replicated for every slave connected to the bus. All communication is controlled by the master, which selects the slave it wishes to communicate with by lowering the appropriate SS line, and then causes a single word (commonly one byte) to be transferred serially to the slave over the signal `MOSI` ("master out, slave in") and simultaneously accepts a single byte from the slave over the signal `MISO` ("master in, slave out"). This transfer is realized by generating 8 clock pulses on the signal `SCK` ("serial clock").

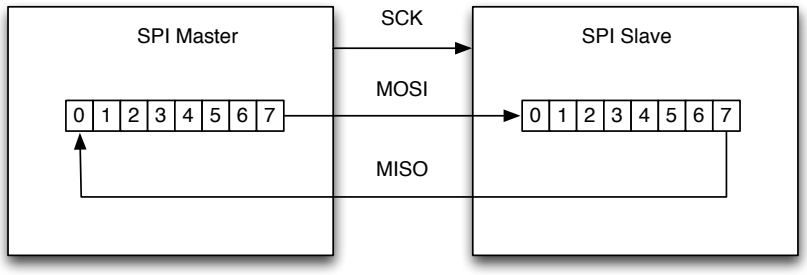



Figure 6.2: Logical Behavior of SPI Bus

The basic data transfer mechanism can be visualized as a pair of linked shift registers as illustrated in Figure 6.2. In this illustration, data are shifted out (in) starting with most significant bit.

The actual protocol behavior is illustrated by the timing diagram in Figure 6.3. In this figure, the master initiates communication with a slave by lowering `SS`. Notice that the `MISO` signal moves from a high-impedance (tri-state) state once the slave is selected. The master controls the transfer with 8 pulses on `SCLK`. In this illustration, data are "clocked" in to the master/slave shift registers on rising clock edges and new data are driven on falling clock edges. Unfortunately, the specific relationship between the clock edges and data is configuration dependent – there are four possible clock modes; however, the LCD requires the illustrated mode (commonly referred to as CPOL=0,CPHA=0).

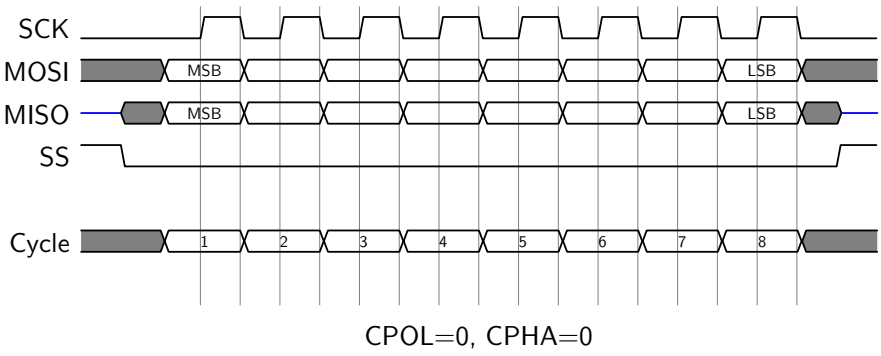



Figure 6.3: SPI Protocol Timing

## 6.2 STM32 SPI Peripheral

```
enum spiSpeed { SPI_SLOW , SPI_MEDIUM, SPI_FAST };

void spiInit(SPI_TypeDef* SPIx);
int spiReadWrite(SPI_TypeDef* SPIx, uint8_t *rbuf,
     const uint8_t *tbuf, int cnt,
     enum spiSpeed speed);
int spiReadWrite16(SPI_TypeDef* SPIx, uint16_t *rbuf,
                   const uint16_t *tbuf, int cnt,
                   enum spiSpeed speed);
```

Listing 6.1: SPI Module Interface

In this chapter we will develop and test a SPI driver with the simple interface illustrated in Listing 6.1. The standard peripheral library defines three SPI devices (SPI1, SPI2, and SPI3) and 8 possible clock prescalers in `stm32f10x_spi.h` – for the 24MHz part on the Discovery board, a prescaler of 8 results in a 3MHz SPI clock (24/8). We use prescalers of 64, 8, and 2 for slow, medium, and fast speeds, respectively. This interface provides for initializing any of these three devices with a relatively generic configuration. There is one data transfer operation which allows exchanging buffers of data with a SPI device. Technically, every data transfer is bidirectional, but many devices do not utilize this capability. Thus, the read/write operations accept null pointers for either send or receive buffers. The SPI device also supports

16-bit transfers, hence our interface provides a 16-bit read/write function. Finally, the interface allows on-the-fly changes of transmission speed. Such a change may be necessary if a bus has two slaves with differing speeds. [1]

Initialization for the SPI module follows the same general sequence required for any peripheral –

1. Enable clocks to peripheral and associated GPIO ports.

2. Configure GPIO pins.

3. Configure the device.

| Function | SPI1 | SPI2 | GPIO configuration |
|---|---|---|---|
| SCK | PA5 | PB13 | Alternate function push-pull (50MHz) |
| MISO | PA6 | PB14 | Input pull-up |
| MOSI | PA7 | PB15 | Alternate function push-pull (50MHz) |

Table 6.1: SPI Pin Configuration

The configuration of the pins required are illustrated in Table 6.1. Not shown are the pins available for hardware control of the slave select line because we implement this through software control. The initialization process is illustrated in Listing 6.2. We are only interested in one mode of operation – the STM32 acting as a master. In more complex systems it may be necessary to significantly modify this procedure, for example passing in a `SPI_InitStructure`. The initialization routine follows the sequence shown above and is easily extended to support additional SPI peripherals.

[1] We use this capability in Chapter 8 where initialization and block transfer speeds differ.

```
static const uint16_t speeds[] = {
  [SPI_SLOW] = SPI_BaudRatePrescaler_64,
  [SPI_MEDIUM] = SPI_BaudRatePrescaler_8,
  [SPI_FAST] = SPI_BaudRatePrescaler_2};

void spiInit(SPI_TypeDef *SPIx)
{
  SPI_InitTypeDef SPI_InitStructure;
  GPIO_InitTypeDef GPIO_InitStructure;

  GPIO_StructInit(&GPIO_InitStructure);
  SPI_StructInit(&SPI_InitStructure);

  if (SPIx == SPI2) {
    /* Enable clocks, configure pins
    ...
    */
    }  else  {
      return;
    }

  SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex;
  SPI_InitStructure.SPI_Mode = SPI_Mode_Master;
  SPI_InitStructure.SPI_DataSize = SPI_DataSize_8b;
  SPI_InitStructure.SPI_CPOL = SPI_CPOL_Low;
  SPI_InitStructure.SPI_CPHA = SPI_CPHA_1Edge;
  SPI_InitStructure.SPI_NSS = SPI_NSS_Soft;
  SPI_InitStructure.SPI_BaudRatePrescaler = speeds[SPI_SLOW];
  SPI_InitStructure.SPI_CRCPolynomial = 7;
  SPI_Init(SPIx, &SPI_InitStructure);

  SPI_Cmd(SPIx, ENABLE);
}
```

Listing 6.2: SPI Initialization

Our basic SPI module supports a single transaction types – read/write which is illustrated in Listing 6.3. The read/write routine iterates over the number of data bytes to be exchanged. Each iteration consists of sending a byte, waiting for the receiver to complete, and then receiving a byte. In the case of the write-only routine, an internal buffer is used to catch and discard a received byte. Similarly, the read-only routine transmits a sequence of 0xff (effectively idle) bytes while receiving. The 16-bit routines work by temporarily modifying the configuration to support 16-bit transfers using the following system calls:

```
SPI_DataSizeConfig(SPIx, SPI_DataSize_16b);
SPI_DataSizeConfig(SPIx, SPI_DataSize_8b);
```

```
int spiReadWrite(SPI_TypeDef* SPIx, uint8_t *rbuf,
         const uint8_t *tbuf, int cnt, enum spiSpeed speed)
{
  int i;

  SPIx->CR1 = (SPIx->CR1 & ~SPI_BaudRatePrescaler_256) |
              speeds[speed];

  for (i = 0; i < cnt; i++){
    if (tbuf) {
      SPI_I2S_SendData(SPIx, *tbuf++);
    } else {
      SPI_I2S_SendData(SPIx, 0xff);
    }
    while (SPI_I2S_GetFlagStatus(SPIx, SPI_I2S_FLAG_RXNE) == RESET);
    if (rbuf) {
      *rbuf++ = SPI_I2S_ReceiveData(SPIx);
    } else {
      SPI_I2S_ReceiveData(SPIx);
    }
  }
  return i;
}
```

Listing 6.3: SPI Read/Write

## 6.3 Testing the SPI Interface

The easiest way to test this SPI interface is by wiring it in "loop-back" mode with MISO directly connected to MOSI and watching data transmissions with the Saleae Logic. Because the Logic expects a select signal (SS) the test program must explicitly configure and control a select signal. In the following section, we will show how to use the SPI interface to control an external EEPROM; for that example we will use PC10 as the select pin (for this loopback example we are using PC3), therefore it is convenient to use the same pin here.

The main routine of a simple test program is illustrated in Listing 6.4. Notice that it tests both 8-bit and 16-bit modes. An fragment of the Logic output for this program is illustrated in Figure 6.4.

```
uint8_t txbuf[4], rxbuf[4];
uint16_t txbuf16[4], rxbuf16[4];

void main()
{
  int i, j;

  csInit(); // Initialize chip select PC03
  spiInit(SPI2);

  for (i = 0; i < 8; i++) {
      for (j = 0; j < 4; j++)
        txbuf[j] = i*4 + j;
      GPIO_WriteBit(GPIOC, GPIO_Pin_3, 0);
      spiReadWrite(SPI2, rxbuf, txbuf, 4, SPI_SLOW);
      GPIO_WriteBit(GPIOC, GPIO_Pin_3, 1);
      for (j = 0; j < 4; j++)
        if (rxbuf[j] != txbuf[j])
          assert_failed(__FILE__, __LINE__);
    }
  for (i = 0; i < 8; i++) {
      for (j = 0; j < 4; j++)
        txbuf16[j] = i*4 + j + (i << 8);
      GPIO_WriteBit(GPIOC, GPIO_Pin_3, 0);
      spiReadWrite16(SPI2, rxbuf16, txbuf16, 4, SPI_SLOW);
      GPIO_WriteBit(GPIOC, GPIO_Pin_3, 1);
      for (j = 0; j < 4; j++)
        if (rxbuf16[j] != txbuf16[j])
          assert_failed(__FILE__, __LINE__);
    }
}
```

Listing 6.4: SPI Loopback Test

### Exercise 6.1 SPI Loopback

Complete the loop back test and capture the resulting output with Saleae. Pay special attention to the "byte order" for 16-bit transfers. Compare this to the byte order for 8-bit transfers. You should use SPI2 and PC3 as the select pin. A template for this project is illustrated in Figure 6.5

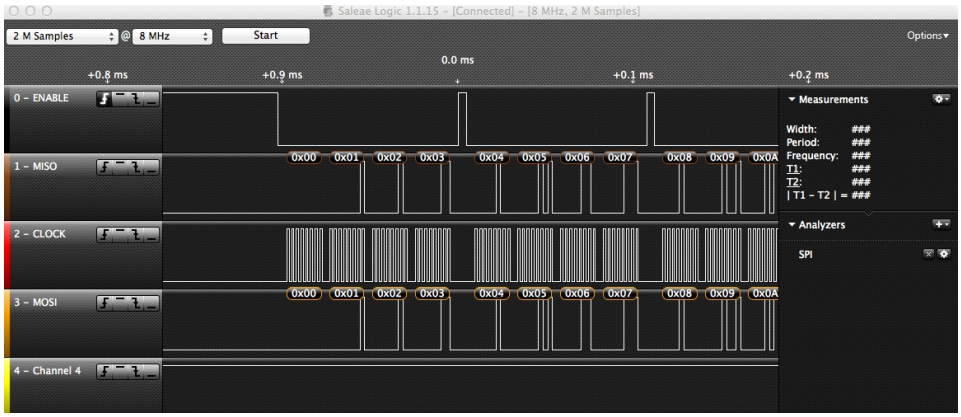



Figure 6.4: SPI Loopback Test Output

```
.../
└── SPIloopback/
    ├── main.c
    ├── Makefile
    ├── spi.c
    └── spi.h
```

Figure 6.5: SPI Loopback Project

## 6.4 EEPROM Interface

One of the simplest and most useful SPI devices is a serial EEPROM (electrically erasable programmable memory) which is often used in embedded devices as a small non-volatile store for configuration parameters and persistent state (e.g. high scores in a game). Data retention for EEPROMs is measured in decades. One limitation is that they can survive only a limited number of write cycles (perhaps a million) and hence should be used for relatively slowly changing information. SPI EEPROMs are available from many manufacturers with common interfaces. The capacities vary from $< 1K$ bit to $> 1M$ bit.

Our choice of an EEPROM for the first SPI example application was dictated by one consideration – the basic operation is intuitive (read/write) and hence it is relatively easy to determine if interface code is working correctly. In the preceding section we used data loop back and the Saleae Logic to ensure that our SPI interface code was operating as expected. It's time to

use that code to build a simple application.

In the following discussion we use a member of the Microchip 25LC160 which is part of a series of devices ranging from 1Kbit for $0.50 (25x010) to 1Mbit for $3.50 (25x1024). For this example, the actual size is irrelevant. We use the common PDIP package illustrated in Figure 6.6.

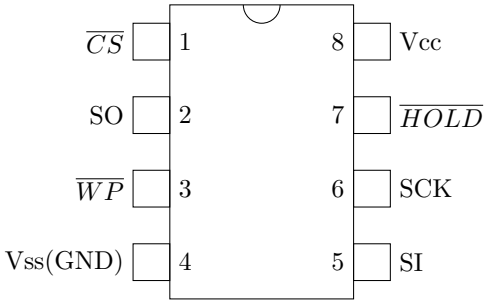



Figure 6.6: PDIP Pinout

The 25xxxx devices implement a simple message protocol over the SPI bus where every transaction consists of one or more instruction bytes sent to the device followed by one or more data bytes either to or from the device. [6] For example, The EEPROM contains a status register which can be used to set various protection modes as well as determine whether the device is busy – EEPROM devices take a long time to complete write operations ! The status read transaction is illustrated in Figure 6.7 – not shown is the SPI clock which completes 16 cycles during this transaction. The master (STM32) selects the EEPROM by lowering SS, it then transmits the 8-bit RDSR code (0x05) and receives the 8-bit status value.

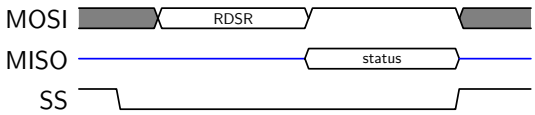



Figure 6.7: Read Status

A command to implement status read is easily implemented:

```
uint8_t eepromReadStatus() {
  uint8_t cmd[] = {cmdRDSR, 0xff};
  uint8_t res[2];
  GPIO_WriteBit(EEPROM_PORT, EEPROM_CS, 0);
  spiReadWrite(EEPROM_SPI, res, cmd, 2, EEPROM_SPEED);
  GPIO_WriteBit(EEPROM_PORT, EEPROM_CS, 1);
  return res[1];
}
```

The status register format is illustrated in Figure 6.8. There are only two bits that concern us at present – WIP (write in progress) and WEL (write enable latch). BP[1:0] are block protect bits which, if set, protect portions of the EEPROM from writes. The WIP bit is especially important – if it is set, then a write is in progress and all commands other than RDSR will fail. As we'll see, it is essential to wait for this bit to be cleared before attempting any other action. The WEL bit must be set in order to perform a data write and it is automatically reset by any write – later we'll see how to set this bit.

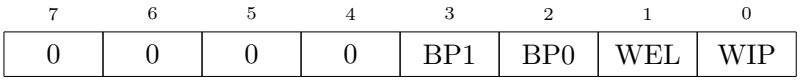



Figure 6.8: EEPROM Status Register

| Instruction | Description | Code |
|---|---|---|
| WRSR | Write Status Register | 0x01 |
| WRITE | Data Write | 0x02 |
| READ | Data Read | 0x03 |
| WRDI | Write Disable | 0x04 |
| RDSR | Read Status Register | 0x05 |
| WREN | Write Enable | 0x06 |

Table 6.2: EEPROM Instructions

.

The full set of available commands are illustrated in Table 6.2. Which we encode as:

```
enum eepromCMD   { cmdREAD  = 0x03, cmdWRITE = 0x02
                   cmdWREN  = 0x06, cmdWRDI  = 0x04,
                   cmdRDSR  = 0x05, cmdWRSR  = 0x01 };
```

The simplest commands are WRDI (Write disable) and WREN (Write enable) which clear and set the WEL bit of the status register respectively.

Each is implemented by writing a single byte – the command – to the EEPROM. No data is returned. Remember, both of these commands will fail if the EEPROM is busy. For example, we implement the an EEPROM enable as follows (notice the status register polling !):

```
#define WIP(x) ((x) & 1)

void eepromWriteEnable(){
  uint8_t cmd = cmdWREN;

  while (WIP(eepromReadStatus()));

  GPIO_WriteBit(EEPROM_PORT, EEPROM_CS, 0);
  spiReadWrite(EEPROM_SPI, 0, &cmd, 1, EEPROM_SPEED);
  GPIO_WriteBit(EEPROM_PORT, EEPROM_CS, 1);
}
```

The most difficult instructions to implement are read and write – both can read or write multiple bytes. It is important to note that these operations must behave differently for various size EEPROMS. For the larger EEPROMS, such as the 25LC160, the address is transmitted as a pair of successive bytes, while for the 25AA040, the most significant address bit (8) is encoded with the command. Here we assume a 16-bit address as illustrated by the read operation in Figure 6.9. A read operation can access any number of bytes – including the entire EEPROM. The operation begins with the instruction code, two address bytes (most significant byte first !) and then 1 or more read operations. The address transmission is most conveniently treated as a 16-bit transfer – if performed with two 8-bit transfers the address will need to have its bytes swapped.

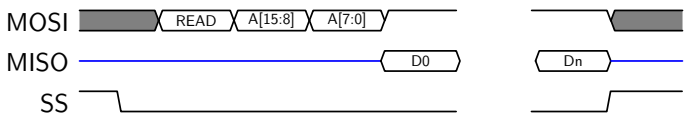



Figure 6.9: EEPROM Read

The write operation is more complicated because of the architecture of an EEPROM. The memory array is organized into “pages” and it is possible to write any or all bytes in a page in a single operation (it is way more efficient to write whole pages simultaneously). The page size is device specific – the

25LC160 has 16 byte pages while larger ones may have 64 byte pages. It is not possible to write bytes in more than one page in a single operation. In practice any bytes written beyond the page selected by the address bits A[16:5] will wrap around within the page. Thus, the eeprom write code must check for page boundaries. The easiest implementation is to return a count of bytes written and limit those bytes to a single page. Some devices limit reads to page boundaries as well – it's important to read the data sheet for the device you plan to use !

```
void eepromInit();
void eepromWriteEnable();
void eepromWriteDisable();
uint8_t eepromReadStatus();
void eepromWriteStatus(uint8_t status);
int eepromWrite(uint8_t *buf, uint8_t cnt, uint16_t offset);
int eepromRead(uint8_t *buf, uint8_t cnt, uint16_t offset);
```

Listing 6.5: EEPROM Module

### Exercise 6.2 Write and Test an EEPROM Module

Implement an eeprom module with the interface illustrated in Listing 6.5. The connection for the eeprom chip are given in Table 6.3. You should use SPI2 at the slow speed (most EEPROMS are fairly slow). The use of a logic analyzer will be essential to debug any issues that arise.

Write a program to perform a test of reading/writing individual locations as well as block read/write.

| EEPROM Pin | EEPROM Signal | STM32 Pin |
|---|---|---|
| 1 | $\overline{CS}$ | PC10 |
| 2 | $SO$ | PB14 |
| 3 | $\overline{WP}$ | 3V3 |
| 4 | $VSS$ | GND |
| 5 | $SI$ | PB15 |
| 6 | $SCK$ | PB13 |
| 7 | $\overline{HOLD}$ | 3V3 |
| 8 | $VCC$ | 3V3 |

Table 6.3: EEPROM Connections

# Chapter 7

# SPI : LCD Display

In this chapter we consider the use of the STM32 SPI interface to communicate with an LCD display module. The LCD is a 1.8" TFT display with 128x160 18-bit color pixels supported by a common (ST7735R) driver chip.

## 7.1 Color LCD Module

The LCD module we consider uses the ST7735R controller.[1]. We refer to the display as a 7735 LCD. The 7735 LCD is a pixel addressable display; every pixel requires multiple bytes to define color – the internal display memory uses 18-bits/pixel (6 bits each for red, blue, and green). We will use this display in a 16-bit mode with 5 bits defining red, 6 bits defining green, and 5 bits defining blue. The display controller automatically extrapolates from 16 bits to 18 bits when pixels are written to the display. This color model is illustrated in Figure 7.1. There are two parts to this figure, the layout of the separate colors within a 16-bit word, and colors resulting from various 16-bit constants.

To understand the required interface to the 7735 LCD, consider the model in Figure 7.1. There are three major components to consider – the controller, the LCD panel, and the display RAM. The display ram contains 18 bits of color information for each of the (128 x 160) pixels in the panel. [2] The data in the display RAM is continuously transferred to the panel – we configure the device to sweep from bottom to top. The model for writing pixel (color) data is somewhat indirect. First, using a separate "control"

[1] There are two variants of the ST7735 – when initializing the controller it is important to do this for the correct variant !

[2] Actually 132x162, but we configure the controller for the smaller number.

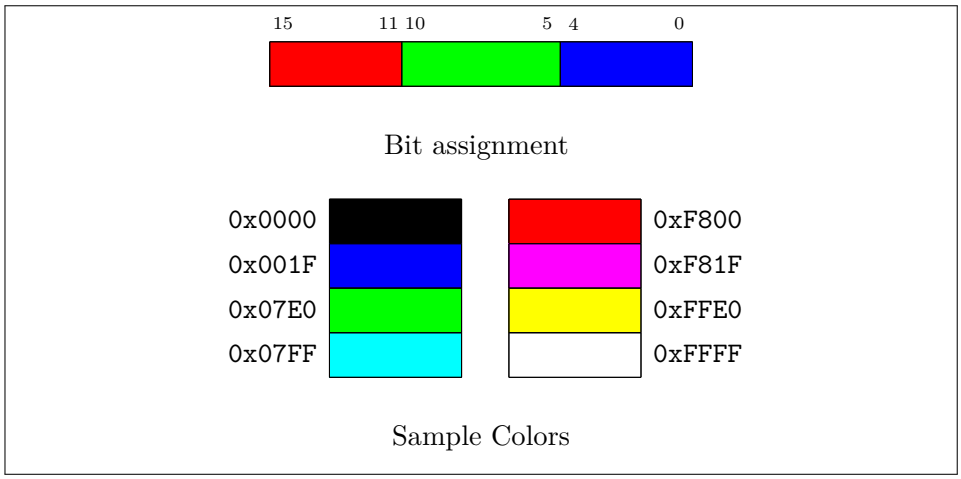



Figure 7.1: Color Encoding

interface, a drawing rectangle is configured. Then, pixel (color) data are written. The location to which the data are written is determined by a pair internal address counters RAC (for row address counter) and CAC (for column address counter). Each successive pixel write causes these address counters to be updated. The 7735 can be configured to "sweep" this rectangle an any left/right top/down order. There are three internal control bits (which we expose in the interface described subsequently). In addition to sweep order, it is possible to "exchange" row and column address and thus support a "landscape" mode. By default we use the mode 0x6 (MY = 1, MX = 1, MV = 0). These are described fully in the ST7735 data manual. [11]

MY Row address order: 1 (bottom to top), 0 (top to bottom)

MX Column address order: 1 (right to left), 0 (left to right)

MV Column/row exchange: 1 (landscape mode), 0 (portrait mode)

As mentioned previously, we configure the panel to accept 16-bit color, so each data write consists of a pair of bytes sent of the SPI interface. These 16-bit color data are automatically translated to 18-bits through an internal lookup table (LUT); the actual display ram pixels are 18-bits.

The 7735 has a separate control signal to differentiate "control" information from "data". Control information (to be discussed subsequently) is

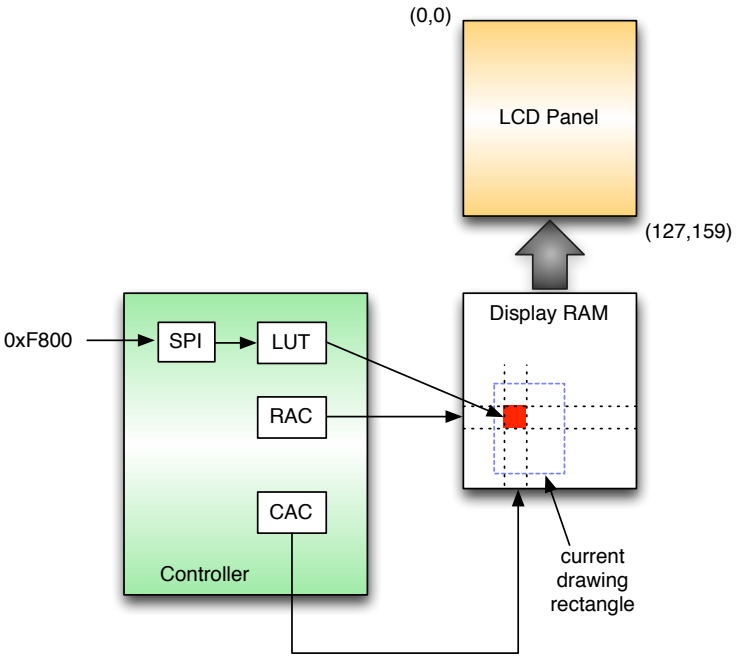



Figure 7.2: Model of 7735 Display

used to configure the display and to set a current drawing rectangle. The advantage to this approach is that once a drawing rectangle is configured, data can be sent in a burst without any additional control information.

A simple "driver" for the 7735 LCD requires only three routines – one to initialize the controller, one to set the drawing rectangle, and one to write color data to the current rectangle. Notice that this interface sets the drawing direction when setting the rectangle (as described above, the `madctl` value of 0x6 corresponds to a top-down/left-right sweep of the drawing rectangle. A `madctl` value of 0x02 corresponds to a bottom-up/left-right sweep and is useful for displaying BMP image files where the data are stored in bottom-up order. A separate routine can be added to control the brightness of the backlight – for now we implement this as on/off. Later, in Chapter 10 we show how to control the brightness with a PWM signal.

```
#define MADCTLGRAPHICS 0x6
#define MADCTLBMP      0x2

#define ST7735_width  128
#define ST7735_height 160

void ST7735_setAddrWindow(uint16_t x0, uint16_t y0,
        uint16_t x1, uint16_t y1, uint8_t madctl);
void ST7735_pushColor(uint16_t *color, int cnt);
void ST7735_init();
void ST7735_backLight(uint8_t on);
```

The following example fills the 7735 screen with a single background color.

```
void fillScreen(uint16_t color)
{
  uint8_t x,y;
  ST7735_setAddrWindow(0, 0, ST7735_width-1, ST7735_height-1,
      ↪MADCTLGRAPHICS);
  for (x=0; x < ST7735_width; x++) {
    for (y=0; y < ST7735_height; y++) {
      ST7735_pushColor(&color,1);
    }
  }
}
```

While this interface can be made more sophisticated, it is sufficient to display text, images, and graphics with sufficient programmer ingenuity.

One complication that must be dealt with is endianess. The STM32 memory is little-endian meaning that a 16-bit quantity is stored in successive memory locations with the low-order byte (bits 0-7) at the lower memory address. In contrast, the 7735 interface assumes that data are received in big-endian order (high-order byte first). Therefore, it is essential that color data are transmitted using 16-bit transfer functions; transferring blocks of color data with 8-bit transfers will result in the high and low order bytes being swapped !

Implementation of basic functionality for the ST7735 relies upon a small set of internal primitives to provide access to the SPI interface and handle "chip select" control. These are illustrated in Listing 7.1. There are two data commands for 8-bit and 16-bit transfers, and a control command. The public interface commands are illustrated in Listing 7.2. Constants such as `ST7735_CASET` (column address set) and `ST7735_RASET` (row address set) are

defined in the ST7735 datasheet [11], although our code is derived from code available at `https://github.com/adafruit/Adafruit-ST7735-Library.git`.

```
static void LcdWrite(char dc, const char *data, int nbytes)
{
  GPIO_WriteBit(LCD_PORT,GPIO_PIN_DC, dc);  // dc 1 = data, 0 =
      ↪control
  GPIO_ResetBits(LCD_PORT,GPIO_PIN_SCE);
  spiReadWrite(SPILCD, 0, data, nbytes, LCDSPEED);
  GPIO_SetBits(LCD_PORT,GPIO_PIN_SCE);
}

static void LcdWrite16(char dc, const uint16_t *data, int cnt)
{
  GPIO_WriteBit(LCD_PORT,GPIO_PIN_DC, dc);  // dc 1 = data, 0 =
      ↪control
  GPIO_ResetBits(LCD_PORT,GPIO_PIN_SCE);
  spiReadWrite16(SPILCD, 0, data, cnt, LCDSPEED);
  GPIO_SetBits(LCD_PORT,GPIO_PIN_SCE);
}

static void ST7735_writeCmd(uint8_t c)
{
  LcdWrite(LCD_C, &c, 1);
}
```

Listing 7.1: ST7735 Internal Primitives

Initialization of the ST7735 is somewhat complex because of the need to initialize internal registers in addition to pins and clock sources. The sequence of initialization commands is best copied from existing software (be careful to use ST7735R initialization code !). The basic initialization process requires sending a series of commands interspersed with delays. We defined a data structure to hold these initialization steps (illustrated in Listing 7.3). The actual initialization code is shown in Listing 7.4. Missing details can be gleaned from the library code from which our code is derived – the elided initialization commands are shown in Listing 7.5.

Wiring for the LCD module is relatively simple as illustrated in Figure 7.1 – the chip select signal for the SD card is left disconnected at this time. Table 7.1 summarizes the necessary connections (constant names used in the example code are shown in parenthesis).

### Exercise 7.1 Complete Interface Code

| TFT Pin | STM32 Pin | Function |
|---|---|---|
| VCC | 5V | Power – 5 Volts |
| BKL | PA1 | Backlight control (GPIO_PIN_BKL) |
| RESET | PC1 | LCD Reset (GPIO_PIN_RST) |
| RS | PC2 | Data/Control (GPIO_PIN_DC) |
| MISO | PB14 | SPI2 MISO |
| MOSI | PB15 | SPI2 MOSI |
| SCLK | PB13 | SPI2 CLK |
| LCD CS | PC0 | LCD select (GPIO_PIN_SCE) |
| SD_CS | PC6 | SD card select |
| GND | GND | Ground |

Figure 7.3: TFT Pin Assignment

Complete the code for the ST7335 driver by examining the reference code at
`https://github.com/adafruit/Adafruit-ST7735-Library`. You'll need to complete the initialization code data structure. To test your code, write a program that displays the three primary colors in a cycle, with an appropriate delay.

### Exercise 7.2 Display Text

An important use for an LCD is to display log messages from your code. The first requirement is to display characters. The reference code cited above includes a simple bit-mapped font glcdfont.c which defines the ASCII characters as 5x7 bit maps (each character is placed in a 6x10 rectangle leaving space between lines (3 pixels) and characters (1 pixel). A fragment of this font is illustrated in Figure 7.1. ASCII 0 is a NULL character and hence not displayed. Many of the low-numbered ASCII characters are unprintable and hence either left blank or filled with a default glyph. Consider the character for 'A' (ASCII 65) also illustrated.

Write a routine to display a single character at a specific location defined by the upper left corner of the character – remember that (0,0) is the upper left corner of the display and (127,159) is the lower right corner. Writing a character requires writing a foreground color to each "on" pixel and a background color to each "off" pixel. Each character should occupy a 6x10 region. It is much faster to write a block to the LCD than one pixel at a time (especially when we introduce DMA.

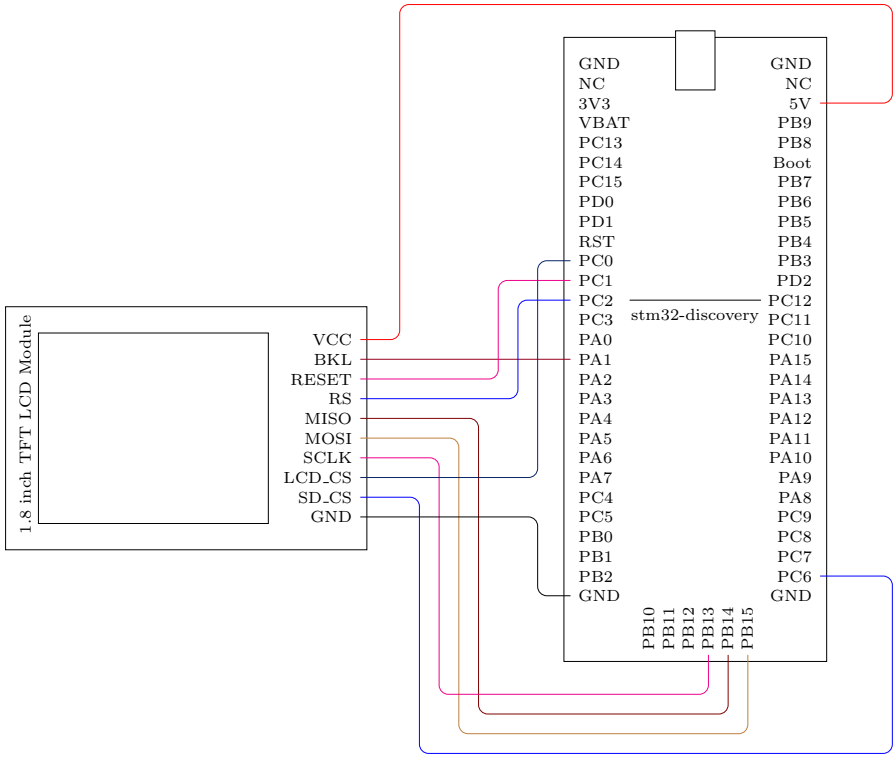



Figure 7.4: Wiring for TFT Module

Extend your routine to support writing lines of text to the screen - consider how you might handle wrap.

### Exercise 7.3 Graphics

Write routines to draw lines and circles of various sizes and colors.

```
#include <stdint.h>

const uint8_t ASCII[] = {
  0x00, 0x00, 0x00, 0x00, 0x00,  // 0
  0x3E, 0x5B, 0x4F, 0x5B, 0x3E,  // 1
  ...
 0x7C, 0x12, 0x11, 0x12, 0x7C,  // 65 A
 0x7F, 0x49, 0x49, 0x49, 0x36,  // 66 B
 0x3E, 0x41, 0x41, 0x41, 0x22,  // 67 C
  ...
```

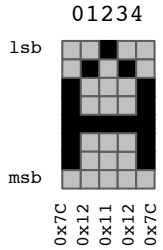



Figure 7.5: Font Fragment

```
#define LOW  0
#define HIGH 1

#define LCD_C LOW
#define LCD_D HIGH

#define ST7735_CASET 0x2A
#define ST7735_RASET 0x2B
#define ST7735_MADCTL 0x36
#define ST7735_RAMWR 0x2C
#define ST7735_RAMRD 0x2E
#define ST7735_COLMOD 0x3A

#define MADVAL(x) (((x) << 5) | 8)
static uint8_t madctlcurrent = MADVAL(MADCTLGRAPHICS);

void ST7735_setAddrWindow(uint16_t x0, uint16_t y0,
        uint16_t x1, uint16_t y1, uint8_t madctl)
{
  madctl = MADVAL(madctl);
  if (madctl != madctlcurrent){
      ST7735_writeCmd(ST7735_MADCTL);
      LcdWrite(LCD_D, &madctl, 1);
      madctlcurrent = madctl;
  }
  ST7735_writeCmd(ST7735_CASET);
  LcdWrite16(LCD_D, &x0, 1);
  LcdWrite16(LCD_D, &x1, 1);

  ST7735_writeCmd(ST7735_RASET);
  LcdWrite16(LCD_D, &y0, 1);
  LcdWrite16(LCD_D, &y1, 1);

  ST7735_writeCmd(ST7735_RAMWR);
}

void ST7735_pushColor(uint16_t *color, int cnt)
{
  LcdWrite16(LCD_D, color, cnt);
}

void ST7735_backLight(uint8_t on)
{
  if (on)
    GPIO_WriteBit(LCD_PORT_BKL,GPIO_PIN_BKL, LOW);
  else
    GPIO_WriteBit(LCD_PORT_BKL,GPIO_PIN_BKL, HIGH);
}
```

Listing 7.2: ST7735 Interface

```
struct ST7735_cmdBuf {
  uint8_t command;   // ST7735 command byte
  uint8_t delay;     // ms delay after
  uint8_t len;       // length of parameter data
  uint8_t data[16];  // parameter data
};

static const struct ST7735_cmdBuf initializers[] = {
  // SWRESET Software reset
  { 0x01, 150, 0, 0},
  // SLPOUT Leave sleep mode
  { 0x11,  150, 0, 0},
  // FRMCTR1, FRMCTR2 Frame Rate configuration -- Normal mode, idle
  // frame rate = fosc / (1 x 2 + 40) * (LINE + 2C + 2D)
  { 0xB1, 0, 3, { 0x01, 0x2C, 0x2D }},
  { 0xB2, 0, 3, { 0x01, 0x2C, 0x2D }},
  // FRMCTR3 Frame Rate configureation -- partial mode
  { 0xB3, 0, 6, { 0x01, 0x2C, 0x2D, 0x01, 0x2C, 0x2D }},
  // INVCTR Display inversion (no inversion)
  { 0xB4,  0, 1, { 0x07 }},
  /* ... */
```

Listing 7.3: ST7735 Initialization Commands (Abbreviated)

```
void ST7735_init()
{
  GPIO_InitTypeDef GPIO_InitStructure;
  const struct ST7735_cmdBuf *cmd;

  // set up pins
  /* ... */

  // set cs, reset low

  GPIO_WriteBit(LCD_PORT,GPIO_PIN_SCE, HIGH);
  GPIO_WriteBit(LCD_PORT,GPIO_PIN_RST, HIGH);
  Delay(10);
  GPIO_WriteBit(LCD_PORT,GPIO_PIN_RST, LOW);
  Delay(10);
  GPIO_WriteBit(LCD_PORT,GPIO_PIN_RST, HIGH);
  Delay(10);

  // Send initialization commands to ST7735

  for (cmd = initializers; cmd->command; cmd++)
    {
      LcdWrite(LCD_C, &(cmd->command), 1);
      if (cmd->len)
        LcdWrite(LCD_D, cmd->data, cmd->len);
      if (cmd->delay)
        Delay(cmd->delay);
    }
}
```

Listing 7.4: ST7735 Initialization

## 7.2 Copyright Information

Our code for the ST7335 is derived from a module available from `https://github.com/adafruit/Adafruit-ST7735-Library.git`

The following copyright applies to that code:

```
/***************************************************
  This is a library for the Adafruit 1.8" SPI display.
  This library works with the Adafruit 1.8" TFT Breakout w/SD card
  ----> http://www.adafruit.com/products/358
  as well as Adafruit raw 1.8" TFT display
  ----> http://www.adafruit.com/products/618

  Check out the links above for our tutorials and wiring diagrams
  These displays use SPI to communicate, 4 or 5 pins are required to
  interface (RST is optional)
  Adafruit invests time and resources providing this open source code,
  please support Adafruit and open-source hardware by purchasing
  products from Adafruit!

  Written by Limor Fried/Ladyada for Adafruit Industries.
  MIT license, all text above must be included in any redistribution
 ****************************************************/
```

## 7.3 Initialization Commands (Remainder)

```
  // PWCTR1 Power control -4.6V, Auto mode
  { 0xC0,  0, 3, { 0xA2, 0x02, 0x84}},
  // PWCTR2 Power control VGH25 2.4C, VGSEL -10, VGH = 3 * AVDD
  { 0xC1,  0, 1, { 0xC5}},
  // PWCTR3 Power control, opamp current smal, boost frequency
  { 0xC2,  0, 2, { 0x0A, 0x00 }},
  // PWCTR4 Power control, BLK/2, opamp current small and medium low
  { 0xC3,  0, 2, { 0x8A, 0x2A}},
  // PWRCTR5, VMCTR1 Power control
  { 0xC4,  0, 2, { 0x8A, 0xEE}},
  { 0xC5,  0, 1, { 0x0E }},
  // INVOFF Don't invert display
  { 0x20,  0, 0, 0},
  // Memory access directions. row address/col address, bottom to
      ↪top refesh (10.1.27)
  { ST7735_MADCTL,  0, 1, {MADVAL(MADCTLGRAPHICS)}},
  // Color mode 16 bit (10.1.30
  { ST7735_COLMOD,   0, 1, {0x05}},
  // Column address set 0..127
  { ST7735_CASET,   0, 4, {0x00, 0x00, 0x00, 0x7F }},
  // Row address set 0..159
  { ST7735_RASET,   0, 4, {0x00, 0x00, 0x00, 0x9F }},
  // GMCTRP1 Gamma correction
  { 0xE0, 0, 16, {0x02, 0x1C, 0x07, 0x12, 0x37, 0x32, 0x29, 0x2D,
          0x29, 0x25, 0x2B, 0x39, 0x00, 0x01, 0x03, 0x10 }},
  // GMCTRP2 Gamma Polarity corrction
  { 0xE1, 0, 16, {0x03, 0x1d, 0x07, 0x06, 0x2E, 0x2C, 0x29, 0x2D,
          0x2E, 0x2E, 0x37, 0x3F, 0x00, 0x00, 0x02, 0x10 }},
  // DISPON Display on
  { 0x29, 100, 0, 0},
  // NORON Normal on
  { 0x13,  10, 0, 0},
  // End
  { 0, 0, 0, 0}
};
```

Listing 7.5: ST7735 Initialization Commands (Remainder)

# Chapter 8

# SD Memory Cards

In this chapter we show how to interface a commodity SD memory card to the STM32 VL Discovery board using the SPI peripheral discussed in Chapter 6. While communicating with an SD memory card is a simple extension of the previously presented work, controlling the card and interpreting the data communicated requires a significant additional software. Fortunately, much of the required software is available in the widely used FatFs module [3]. Only a modest amount of porting is required to utilize this module with our SPI driver.

The STM32 processors on the VL discovery board are relatively memory constrained – 128K bytes flash and 8K bytes ram – which limits the ability to store large quantities of data either as inputs to or outputs from an embedded program. For example, in a game application it might be desirable to access sound and graphic files or in a data logging application, to store extended amounts of data. Furthermore, accessing the contents of the STM32 flash requires a special interface and software. In such applications it is desirable to provide external storage which the STM32 can access while running and the user/programmer can easily access at other times. Commodity flash memory cards (in particular SD cards) provide a cost effective solution which can reasonably easily be accessed by both the processor and user. In practice, these cards have file systems (typically FAT) and can be inserted in commonly available adaptors to be accessed by a desktop machine. Furthermore, the physical interface has a SPI mode which is accessible using the code described in Chapter 6.

Physically, SD memory cards consist of a flash memory array and a control processor which communicates with a host over either the SD bus (a parallel bus) or the SPI bus. Communication is transaction based – the host

sends a command message to the SD card and receives a response. Access to the flash memory of an SD card is performed through fixed-sized block reads and writes which are also implemented with the command protocol. A basic overview is provided in [4].

The data on an SD card is organized as a file system – cards below 2GB are typically formatted as FAT16 file systems. In a FAT file system, the first several storage blocks are used to maintain data about the file system – for example allocation tables – while the remaining blocks are used to store the contents of files and directories.

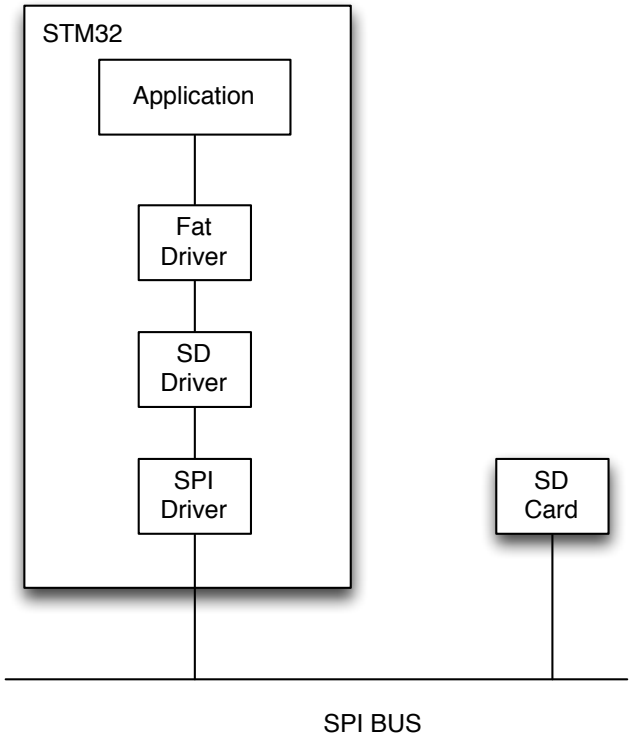



Figure 8.1: SD Card Software Stack

Although we have previously developed a SPI bus driver that is capable of communicating with an SD card, we are missing several key software components. Consider Figure 8 which illustrates the necessary software stack. An application, which wishes to access data stored on an SD Card, utilizes file

level commands such as open, read, and write to access specific files within the SD card file system. These commands are provided by a FAT file system driver. The FAT file system issues commands at the level of block reads and writes without any knowledge of how these commands are implemented. A separate SD driver implements these commands. Finally, the SD driver utilizes the SPI interface to communicate with the SD Card.

Fortunately, it is not necessary to write all of this software. In this chapter we will describe the use of the FatFs generic file system. [3] This open source package provides most of the components required including a "generic" SD driver that is relatively easily modified to utilize the SPI driver presented in Chapter 6.

To understand how an application interacts with FatFs, consider the example derived from the FatFs sample distribution illustrated in Listing 8.1. This example fragment assumes it is communicating with an SD card formatted with a fat file system which contains file in the root directory called `MESSAGE.TXT`. The program reads this file, and creates another called `HELLO.TXT`. Notice the use of relatively standard file system commands.

```
f_mount(0, &Fatfs);/* Register volume work area */

xprintf("\nOpen an existing file (message.txt).\n");
rc = f_open(&Fil, "MESSAGE.TXT", FA_READ);

if (!rc) {
  xprintf("\nType the file content.\n");
  for (;;) {
    /* Read a chunk of file */
    rc = f_read(&Fil, Buff, sizeof Buff, &br);
    if (rc || !br) break;/* Error or end of file */
    for (i = 0; i < br; i++)/* Type the data */
      myputchar(Buff[i]);
  }
  if (rc) die(rc);
  xprintf("\nClose the file.\n");
  rc = f_close(&Fil);
  if (rc) die(rc);
}

xprintf("\nCreate a new file (hello.txt).\n");
rc = f_open(&Fil, "HELLO.TXT", FA_WRITE | FA_CREATE_ALWAYS);
if (rc) die(rc);

xprintf("\nWrite a text data. (Hello world!)\n");
rc = f_write(&Fil, "Hello world!\r\n", 14, &bw);
if (rc) die(rc);
xprintf("%u bytes written.\n", bw);
```

Listing 8.1: FatFs Example

## 8.1 FatFs Organization

The following discussion refers to the current (0.9) version of FatFs. The code distribution is organized as illustrated in Figure 8.2

The interface between the fat file system driver and the SD driver is defined in the module `diskio.h` illustrated in Listing 8.2. We have modified the default distribution to include meaningful parameter names. An initialized disk can be read, written, and controlled (ioctl). The read/write commands are restricted to the block level (the size of blocks depends upon the SD card). The ioctl function provides a means to determine the SD card "geometry" (e.g. number and size of blocks), and may provide additionally functionality to enable power control. The low level implementation described in the sequel only supports ioctl functions to determine the disk "geometry" and to force

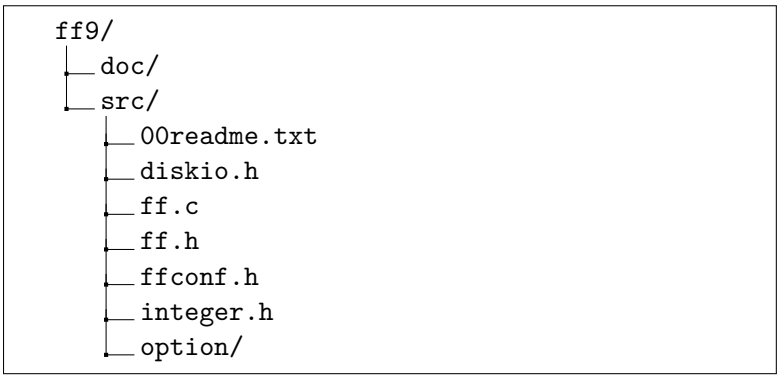



Figure 8.2: FatFs Distribution Organization

completion of pending writes.

```
/*---------------------------------------*/
/* Prototypes for disk control functions */

int assign_drives (int, int);
DSTATUS disk_initialize (BYTE drv);
DSTATUS disk_status (BYTE drv);
DRESULT disk_read (BYTE drv, BYTE* buff, DWORD sector, BYTE count);
#if _READONLY == 0
DRESULT disk_write (BYTE drv, const BYTE* buff, DWORD sector, BYTE
    ↪count);
#endif
DRESULT disk_ioctl (BYTE drv, BYTE ctl, void* buff);
```

Listing 8.2: Low Level Driver Interface

## 8.2 SD Driver

As described previously, the SD driver implements five functions to support the FatFs and uses the SPI driver to communicate with the SDCard. The SDCard communication protocol is transaction based. Each transaction begins with a one-byte command code, optionally followed by parameters, and then a data transfer (read or write). The SDCard protocol is well documented. [9]. In the following we present a few examples to illustrated the basic concepts. Fortunately, it is not necessary to create this module from scratch. There is a distribution of sample projects http://elm-chan.org/fsw/ff/

`ffsample.zip` – we use the generic example. Alternatively, there is a more sophisticated port to the STM32 (`http://www.siwawi.arubi.uni-kl.de/avr_projects/arm_projects/arm_memcards/index.html#stm32_memcard`). The sample code is organized as illustrated in Figure 8.3.

```
ffsample/
├── 00readme.txt
├── lpc17xx/
│   ├── xprintf.h
│   ├── xprintf.c
│   └── ...
├── ...
└── generic/ .3 main.c
    ├── mmcbb.c
    └── ...
```

Figure 8.3: FatFs Sample Code Organization

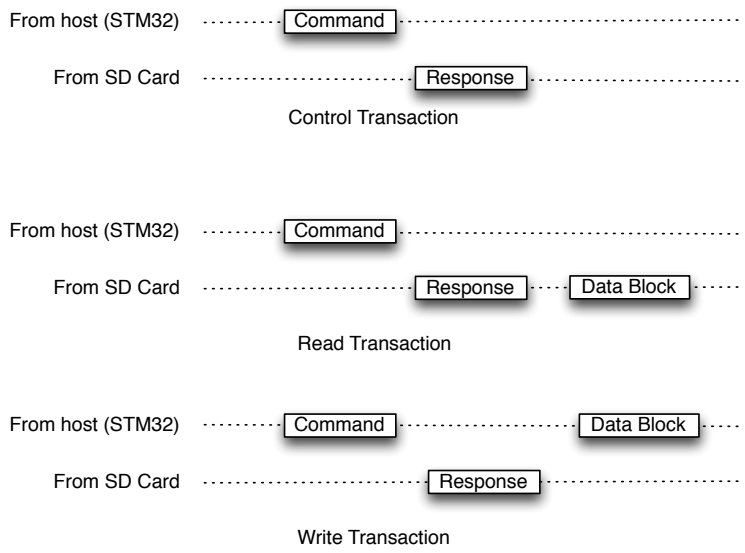



Figure 8.4: SD Transactions

The three basic transaction types that concern us are illustrated in Figure 8.2. Every transaction begins with a command issued by the host

(in this case the STM32) followed by a response from the SD card. For pure control operations (e.g querying the card status or configuration, the SD card response terminates the transactions. For read or write transactions, the response is followed by the transmission of a data block from (write) or to the host (read). There are other cases including multi-block data transactions and error conditions which are not illustrated. It should be clear that these transaction types are sufficient to implement the functionality required by the FatFs.

We will not delve deeply into the format of the information transferred during transactions except to enumerate a few of the commands defined by the SD Card specifications and to point out that all of the fields of a transaction may optionally be protected by CRC codes. A subset of the SD card commands are illustrated in Table 8.2. Notice that there are commands to reset and initialize the card, read/write parameters (e.g. block length), and read/write data blocks. These commands are supported by multiple response formats with lengths that vary from 1-5.

| Command | Description |
|---|---|
| CMD0 | Reset the SD Card |
| CMD1 | Initialize card |
| CMD8 | Write voltage level |
| CMD9 | Request card-specific data (CSD) |
| CMD10 | Request card identification (CID) |
| CMD12 | Stop transmission |
| CMD13 | Request card status |
| CMD16 | Set transfer block length |
| CMD17 | Read single block |
| CMD18 | Read multiple blocks |
| CMD24 | Write single block |
| CMD25 | Write multiple blocks |
| CMD58 | Read OCR register |
| ACMD23 | Number of blocks to erase |
| ACMD41 | Initialize card |

Table 8.1: Some SD Card Commands

Our implementation of the SD driver is a simple port of `generic/mmbc.c` (see Figure 8.3). There are only a small number of routines that must be modified in order to utilize this module. These are presented in Listings 8.3 and 8.4. Additionally, the sample code utilizes a wait function (`DLY_US`) that counts

microseconds while our delay counts milliseconds. It is necessary to make appropriate modifications throughout the code. There is nothing fundamental about most of the delay periods, but any changes should attempt a similar total delay. Finally, we modified the `disk_initialize` routine to set the SPI speed to a slow rate during initialization and a fast rate after initialization is successfully completed.

### Exercise 8.1 FAT File System

Port the generic FatFs driver and example program to the Discovery board. You may use the xprintf functions distributed with the sample code. FatFs is a highly configurable system. The configuration is controlled through `ffconf.h`. For this exercise you should use the default settings. Once your code is working, you may wish to try some of the available options. The basic steps follow – an example Makefile is given in Listing 8.5

1. modify mmcbb.c from the generic example
2. create a project which includes ff.c from the fat file system, mmcbb.c, your spi driver, and the required STM32 library files.
3. format your SD card and create a file "MESSAGE.TXT"

In order to use `xprintf` you will need to include some code in your main

```
#include "xprintf.h"

void myputchar(unsigned char c)
{
  uart_putc(c, USART1);
}

unsigned char mygetchar()
{
  return uart_getc(USART1);
}

int main(void)
{
  ...
  xfunc_in = mygetchar;
  xfunc_out = myputchar;
  ...
}
```

```
#include <stm32f10x.h>
#include <stm32f10x_spi.h>
#include <stm32f10x_rcc.h>
#include <stm32f10x_gpio.h>
#include "spi.h"

#define GPIO_Pin_CS GPIO_Pin_6
#define GPIO_CS GPIOC
#define RCC_APB2Periph_GPIO_CS RCC_APB2Periph_GPIOC
#define SD_SPI SPI2

enum spiSpeed Speed = SPI_SLOW;

void Delay(uint32_t);
/* ... */
/*-----------------------------*/
/* Transmit bytes to the card */
/*-----------------------------*/

static
void xmit_mmc (
  const BYTE* buff, /* Data to be sent */
  UINT bc     /* Number of bytes to send */
)
{
  spiReadWrite(SD_SPI, 0, buff, bc, Speed);
}

/*-----------------------------*/
/* Receive bytes from the card */
/*-----------------------------*/

static
void rcvr_mmc (
  BYTE *buff, /* Pointer to read buffer */
  UINT bc   /* Number of bytes to receive */
)
{
  spiReadWrite(SD_SPI, buff, 0, bc, Speed);
}
```

Listing 8.3: SD Driver Routines

```
/*-----------------------*/
/* Deselect the card     */
/*-----------------------*/

static
void deselect (void)
{
  BYTE d;

  GPIO_SetBits(GPIO_CS, GPIO_Pin_CS);
  rcvr_mmc(&d, 1);  /* Dummy clock (force DO hi-z for multiple
      ↪slave SPI) */
}

/*-------------------*/
/* Select the card   */
/*-------------------*/

static
int select (void) /* 1:OK, 0:Timeout */
{
  BYTE d;

  GPIO_ResetBits(GPIO_CS, GPIO_Pin_CS);
  rcvr_mmc(&d, 1);  /* Dummy clock (force DO enabled) */

  if (wait_ready()) return 1; /* OK */
  deselect();
  return 0;     /* Failed */
}


INIT_PORT()
{
  GPIO_InitTypeDef GPIO_InitStructure;

  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIO_CS, ENABLE);
  /* Configure I/O for Flash Chip select */
  GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_CS;
  GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_Out_PP;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_Init(GPIO_CS, &GPIO_InitStructure);
  deselect();
}
```

Listing 8.4: SD Driver Routines (cont.)

```
TEMPLATEROOT = path_to_template_root

# additional compilation flags

CFLAGS  +=  -g  -Ipath_to_ff9/src
ASFLAGS += -g
LDLIBS  += -lm

# project files

vpath %.c path_to_ff9/src
vpath %.c path_to_ff9/src/option

# ccsbcs.o
OBJS=    $(STARTUP)  main.o
OBJS+=   ff.o spi.o uart.o xprintf.o mmcbb.o
OBJS+=   stm32f10x_gpio.o stm32f10x_rcc.o  stm32f10x_usart.o misc.o
OBJS+=   stm32f10x_spi.o   core_cm3.o

include $(TEMPLATEROOT)/Makefile.common
```

Listing 8.5: Makefile for SD Card Project

## 8.3 FatFs Copyright

```
FatFs module is an open source software to implement FAT file
    ↪system to
small embedded systems. This is a free software and is opened for
    ↪education,
research and commercial developments under license policy of
    ↪following terms.

 Copyright (C) 2011, ChaN, all right reserved.

* The FatFs module is a free software and there is NO WARRANTY.
* No restriction on use. You can use, modify and redistribute it
    ↪for
  personal, non-profit or commercial product UNDER YOUR
* RESPONSIBILITY.
* Redistributions of source code must retain the above copyright
    ↪notice.
```

# Chapter 9

# I$^2$C – Wii Nunchuk

In this chapter we introduce I$^2$C, the third major protocol we use to interface with external modules. I$^2$C is a two wire protocol used to connect one or more "masters" with one or more "slaves", although we only discuss the case of a single master (the STM32) communicating with slave devices. An example configuration is illustrated in Figure 9.1. In this configuration, a single master communicates with several slaves over the pair of signal wires `SDA` and `SCL`. Example slave devices include temperature, humidity, and motion sensors as well as serial EEPROMs.

As we shall see, the software required to interface with I$^2$C devices is considerably more complicated than with SPI. For example, I$^2$C has multiple error conditions that must be handled, SPI has no error conditions at the physical level. Similarly, I$^2$C has multiple transaction types, while SPI has a single basic transaction type. Furthermore, SPI is generally a much faster bus (1-3Mbit/sec vs 100-400Kbit/sec). The greatest advantage of I$^2$C over SPI ist that the number of wires required by I$^2$C is constant (2) regardless of the number of connected devices whereas SPI requires a separate select line for each device. In place of select lines, I$^2$C devices have internal addresses and are selected by a master through the transmission of this address over the bus. This difference makes I$^2$C a good choice where a large number of devices must be connected. Finally, I$^2$C is a symmetric bus which can support multiple masters whereas SPI is completely asymmetric.

The remainder of this chapter is organized as follows. We begin with an introduction to the I$^2$C protocol in Section 9.1. We then discuss the use of I$^2$C to communicate with a Wii Nunchuk. The Wii Nunchuk is an inexpensive input device that includes a joystick, two buttons, and a three axis accelerom-

eter (reported to be an LIS302 from ST Microelectronics [24]). Finally, we present basic $I^2C$ communication module.

## 9.1 $I^2C$ Protocol

In this Section we present a concise overview of the $I^2C$ bus protocol which covers only those aspects of the protocol necessary to understand this chapter. For a more complete description see the $I^2C$ specification manual [7].

Electrically, $I^2C$ is a "wired-or" bus – the value of the two signal wires is "high" unless one of the connected devices pulls the signal low. On the left side of Figure 9.1 are two resistors that force ("pull up") the default value of the two bus wires to VCC (typically 3.3V). Any device on the bus may safely force either wire low (to GND) at any time because the resistors limit the current draw; however, the communication protocol constrains when this should occur. The two wires are called SCL (serial clock line) and SDA (serial data/address). To communicate, a master drives a clock signal on SCL while driving, or allowing a slave to drive SDA. Thus, the bit-rate of a transfer is determined by the master.

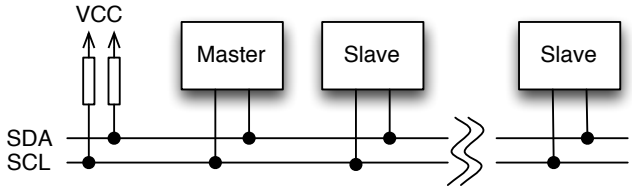



Figure 9.1: Typical $I^2C$ Configuration

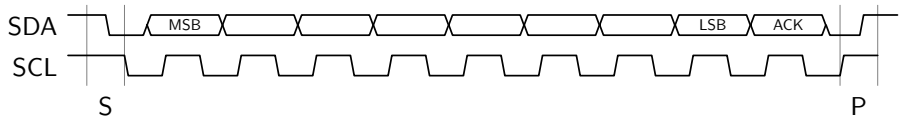
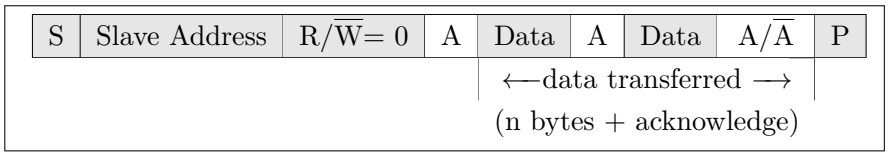
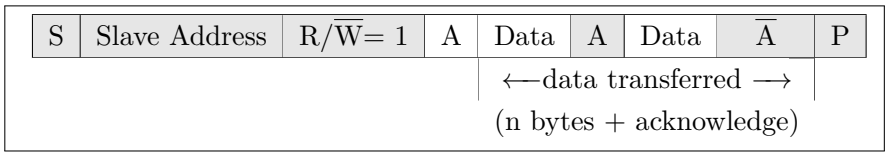



Figure 9.2: $I^2C$ Physical Protocol

Communication between a master and a slave consists of a sequence of transactions where the master utilizes the `SCL` as a clock for serial data




Write Transaction




Read Transaction

From master to slave

From slave to master

A = Acknowledge

$\overline{A}$ = Not Acknowledge

S = Start condition

P = Stop Condition

Figure 9.3: I²C Write and Read Transactions

driven by the master or a slave on `SDA` as illustrated in Figure 9.2. Every transaction begins with a Start condition (S) and ends with Stop condition (P). A transaction consists of a sequence of bytes, delivered most significant bit (MSB) first, each of which is terminated by an Acknowledge (ACK), such as illustrated here, or Not Acknowledge (NACK). The data may be sent by either the slave or the master, as the protocol dictates, and the ACK or NACK is generated by the receiver of the data. Start (S) and Stop (P) conditions are always generated by the master. A high to low transition on SDA while SCL is high defines a Start. A low to high transition on SDA while SCL is high defines a Stop.

There are three types of transactions on the I²C bus, all of which are initiated by the master. These are write, read, and combined transactions, where a combined transaction concatenates a write and read transaction. The first two of these are illustrated in Figure 9.3 – combined transactions are not discussed in this book. Furthermore, there are two addressing modes: 7-bit

addressing, as used in the transactions described in this book, and 10-bit addressing which supports more devices on a single bus at the expensive a more complex transaction format.

All 7-bit address transactions begin with a start event, the transmission of a slave address and a bit which indicates whether this is a write or read transaction by the master. The address phase it terminated by an ACK (0) provided by the slave or NACK (1). Notice that in the event there is no matching slave the electrical properties of the bus guarantee that a NACK is received by the master. In the event that address transmission is followed by a NACK, the master is obliged to generate a stop condition to terminate the transaction thus returning the bus to an idle state – i.e. both signals high.

In a write transaction, the address phase if followed by a series of data bytes (MSB first) transmitted by the master each of which is followed by an ACK or NACK by the slave. The transaction is terminated with a Stop condition after the master has sent as much data as it wishes or the slave has responded with a NACK.

A read transaction differs from a write transaction in that the data are provided by the slave and the ACK/NACK by the master. In a read transaction, the master responds to the last byte it wishes to receive with a NACK. The transaction is terminated with a Stop condition.

The protocol specification describes combined transactions, 10-bit addressing, multi-master buses, as well as details of the physical bus.

## 9.2 Wii Nunchuk

Wii Nunchuks are inexpensive input devices with a joystick, two buttons, and a three-axis accelerometer as illustrated in Figure 9.4. Notice particularly the three axes X, Y, and Z which correspond to the data produced by the accelerometer, joystick . X is right/left, Y is forward/backwards, and Z is up/down (accelerometer only). The $I^2C$ bus is used to initialize the Nunchuk to a known state and then to regularly "poll" its state. There is extensive documentation on the web from which this chapter is drawn (e.g. [8]).

The data are read from the Nun chuck in a six-byte read transaction. These data are formatted as illustrated in Figure 9.5 and are read beginning with byte 0x0 (little-endian). The only complication with this format is that the 10-bit/axis accelerometer data are split.

Communication with the Nunchuk consists of two phases – an initialization phase (executed once) in which specific data are written to to Nunchuk

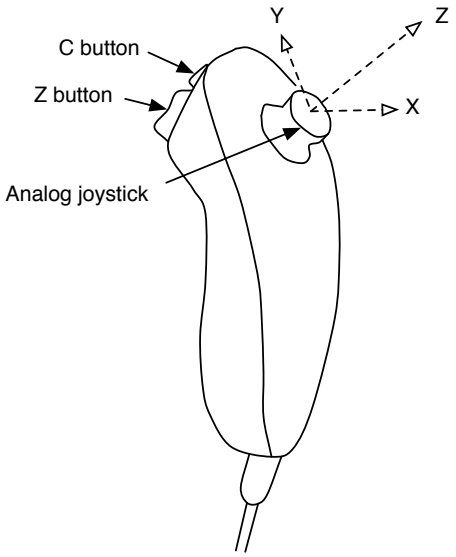



Figure 9.4: Wii Nunchuk

| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| 0x00 | Joystick JX | | | | | | | |
| 0x01 | Joystick JY | | | | | | | |
| 0x02 | Accelerometer AX[9:2] | | | | | | | |
| 0x03 | Accelerometer AY[9:2] | | | | | | | |
| 0x04 | Accelerometer AZ[9:2] | | | | | | | |
| 0x05 | AZ[1:0] | | AY[1:0] | | AX[1:0] | | C | Z |

Figure 9.5: Nunchuk Data

and a repeated read phase in which the six data bytes are read. Each read phase consists of two transactions – a write transaction which sets the read address to zero, and a read transaction. The following code fragments illustrates the initialization phase. The initialization process is described in [23]. Essentially initialization consists of two write transactions, each of which writes a single byte to a register internal to the $I^2C$ slave ( reg[0xf0] = 0x55, reg[0xfb] = 0x00 ). The write routine takes three parameters, the $I^2C$ interface to use

(I2C1 or I2C2), a buffer of data, the buffer length, and the slave address. The I2C transaction routines, discussed in Section 9.3, return error conditions which are ignored in this fragment.

```
// Init

#define NUNCHUK_ADDRESS 0xA4

const uint8_t buf[]  = {0xf0, 0x55};
const uint8_t buf2[] = {0xfb, 0x00};

I2C_Write(I2C1, buf, 2, NUNCHUK_ADDRESS);
I2C_Write(I2C1, buf2, 2, NUNCHUK_ADDRESS);
```

The Nunchuk read process consists of writing a 0 and then reading 6 bytes of data (as described above). Again, we have ignored any error return.

```
// Read

uint8_t data[6];
const uint8_t buf[] = {0};

I2C_Write(I2C1, buf, 1, NUNCHUK_ADDRESS);
I2C_Read(I2C1, data, 6, NUNCHUK_ADDRESS);
```

Reassembly of the data is rather simple; interpretation may be another matter. The joystick data are in the range 0..255 roughly centered at 128 – we found it necessary to calibrate this based upon the value at startup. Furthermore, the dynamic range was somewhat less than the full range (approximately 30-220).

The accelerometer data are in the range 0..1023 where 0 corresponds to -2g and 1023 corresponds to +2g. The accelerometer can be used both to detect motion (acceleration), but also as a "tilt sensor" when it is not in motion because we can use the earth's gravitational field as a reference. [10] Suppose we have measured values of gravity in three dimensions – $G_x, G_y, G_z$ – we know that

$$G_x^2 + G_y^2 + G_z^2 = 1g^2$$

From this, it is possible to compute "pitch" (rotation around the X axis), "roll" (rotation around the Y axis) and "yaw" (rotation around the Z axis). For joystick replacement, it is sufficient to compute (after range conversion to -512..511).

$$pitch = atan\left(\frac{AX}{\sqrt{AY^2 + AZ^2}}\right)$$

and

$$roll = atan\left(\frac{AY}{\sqrt{AX^2 + AZ^2}}\right)$$

Keep in mind that this is for 360 degrees and a reasonable of motion to measure is perhaps 90 degrees.

### Exercise 9.1 Reading Wii Nunchuk

Using the I$^2$C interface code described in Section 9.3 write a program to track the state of the two joystick interface (one based on the accelerometer) of a Wii Nunchuk and display this on the 5110 LCD. Your program should use two letters, x and c, as cursors. When the c (x) button is pressed, the c (x) cursor should be displayed as an upper case letter, otherwise is should be displayed as a lower case letter. You should probably start with an I$^2$C speed of 10000 – the various Nunchuk clones appear to be unreliable at speeds in the range of 100,000.

This is a challenging program to write. You will need to first learn how to communicate with the Nunchuk. Your code will have to appropriately scale the cursor position information to display each cursor in an appropriate location. When working correctly, it should be possible to move each cursor to all for corners of the screen with reasonable motion.

You will need to modify your Makefile to include the standard math libraries by adding the following definition (assuming you are modifying the demo template).

```
LDLIBS += -lm
```

In order to debug your I$^2$C communications it is recommended that you use the Saleae logic to capture any communication events. An example is shown in Figure 9.6. This example illustrates the first phase of reading data from the Nunchuk – writing a 0 to the Nunchuk address. Notice that the start condition is indicated by a green dot and the stop condition is indicated by a red square. Setting up the Saleae logic for the I$^2$C protocol is similar to the serial protocol, but with a different protocol analyzer.

The hardware configuration required is relatively simple. You will need a Nunchuk adaptor as illustrated in Figure 1.7. There are four signals to connect as shown in Table 9.1.

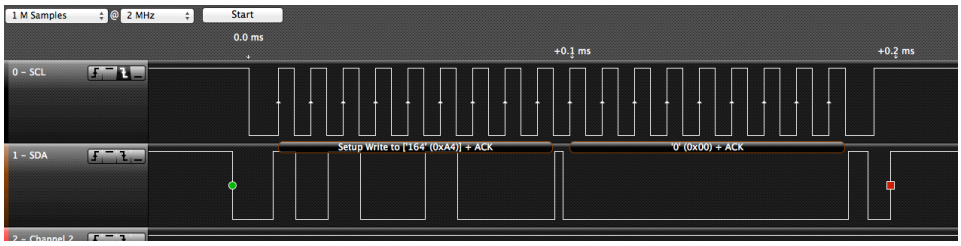



Figure 9.6: Sample Logic Capture for I$^2$C

| | STM32 Interface | |
|---|---|---|
| | I2C1 | I2C2 |
| + | 3V3 | 3V3 |
| - | GND | GND |
| c | PB6 | PB10 |
| d | PB7 | PB11 |

Table 9.1: Connection to Nunchuk Adaptor

## 9.3 STM32 I$^2$C Interface

The STM32 I$^2$C device is extremely complicated. For example, read transactions with 1, 2, and more than 2 bytes are handled in a significantly different manner. The best references are the programmer's manual ([21, 20]) and ST application note AN2824 [13]. The later describes examples for polling, interrupt-driven, and DMA-driven interfaces. Unfortunately, the example code weaves these three cases together and further does not make use of the standard peripheral library. We have rewritten a polling-based solution using the peripheral library.

As with all STM32 devices, the first task is to properly initialize the device including clocks and pins. Our initialization code is illustrated in Listing 9.1.

The write transaction implementation is the simplest of the two transaction types with few special cases. This is illustrated in Listing 9.2. This follows Figure 3 from AN2824. Comments of the form EVn (e.g. EV5) refer to states as described in the programmers manual. The code provided does not attempt to recover from I$^2$C interface errors. Indeed, an interrupt handler is required even to detect all possible errors.

The final transaction we consider is the Read Transaction. As noted in AN2824, there are separate cases required for 1 byte, 2 byte, and greater than 2 byte reads. Furthermore, there are some time critical sections of code which must be executed without interruption ! The shared code is illustrated in Listing 9.3 with the 1, 2, and more than 2 byte cases illustrated in Listings 9.4, 9.5, and 9.6, respectively.

```
void I2C_LowLevel_Init(I2C_TypeDef* I2Cx, int ClockSpeed, int
OwnAddress)
{
    GPIO_InitTypeDef  GPIO_InitStructure;
    I2C_InitTypeDef  I2C_InitStructure;

    // Enable GPIOB clocks

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);

    // Configure I2C clock and GPIO

    GPIO_StructInit(&GPIO_InitStructure);

    if (I2Cx == I2C1){

        /* I2C1 clock enable */

        RCC_APB1PeriphClockCmd(RCC_APB1Periph_I2C1, ENABLE);

        /* I2C1 SDA and SCL configuration */

        GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6 | GPIO_Pin_7;
        GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
        GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_OD;
        GPIO_Init(GPIOB, &GPIO_InitStructure);

        /* I2C1 Reset */

        RCC_APB1PeriphResetCmd(RCC_APB1Periph_I2C1, ENABLE);
        RCC_APB1PeriphResetCmd(RCC_APB1Periph_I2C1, DISABLE);

      }
      else { // I2C2 ...}

  /* Configure I2Cx                */

    I2C_StructInit(&I2C_InitStructure);
    I2C_InitStructure.I2C_Mode = I2C_Mode_I2C;
    I2C_InitStructure.I2C_DutyCycle = I2C_DutyCycle_2;
    I2C_InitStructure.I2C_OwnAddress1 = OwnAddress;
    I2C_InitStructure.I2C_Ack = I2C_Ack_Enable;
    I2C_InitStructure.I2C_AcknowledgedAddress =
    I2C_AcknowledgedAddress_7bit;
    I2C_InitStructure.I2C_ClockSpeed = ClockSpeed;

    I2C_Init(I2Cx, &I2C_InitStructure);
    I2C_Cmd(I2Cx, ENABLE);
}
```

Listing 9.1: Initializing I²C Device

```
#define Timed(x) Timeout = 0xFFFF; while (x) \
 { if (Timeout-- == 0) goto errReturn;}

Status I2C_Write(I2C_TypeDef* I2Cx, const uint8_t* buf,
                 uint32_t nbyte, uint8_t SlaveAddress) {

    __IO uint32_t Timeout = 0;

    if (nbyte)
      {
        Timed(I2C_GetFlagStatus(I2Cx, I2C_FLAG_BUSY));

        // Intiate Start Sequence

        I2C_GenerateSTART(I2Cx, ENABLE);
        Timed(!I2C_CheckEvent(I2Cx, I2C_EVENT_MASTER_MODE_SELECT));

        // Send Address  EV5

        I2C_Send7bitAddress(I2Cx, SlaveAddress,
        I2C_Direction_Transmitter);
        Timed(!I2C_CheckEvent(I2Cx,
        I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED));

        // EV6 Write first byte EV8_1

        I2C_SendData(I2Cx, *buf++);

        while (--nbyte) {

          // wait on BTF

          Timed(!I2C_GetFlagStatus(I2Cx, I2C_FLAG_BTF));
          I2C_SendData(I2Cx, *buf++);
        }

        Timed(!I2C_GetFlagStatus(I2Cx, I2C_FLAG_BTF));
        I2C_GenerateSTOP(I2Cx, ENABLE);
        Timed(I2C_GetFlagStatus(I2C1, I2C_FLAG_STOPF));
      }
    return Success;
 errReturn:
    return Error;
}
```

Listing 9.2: $I^2C$ Write Transaction

```
Status I2C_Read(I2C_TypeDef* I2Cx, uint8_t *buf,
                uint32_t nbyte, uint8_t SlaveAddress) {

  __IO uint32_t Timeout = 0;

  if (!nbyte)
    return Success;

  // Wait for idle I2C interface

  Timed(I2C_GetFlagStatus(I2Cx, I2C_FLAG_BUSY));

  // Enable Acknowledgment, clear POS flag

  I2C_AcknowledgeConfig(I2Cx, ENABLE);
  I2C_NACKPositionConfig(I2Cx, I2C_NACKPosition_Current);

  // Intiate Start Sequence (wait for EV5)

  I2C_GenerateSTART(I2Cx, ENABLE);
  Timed(!I2C_CheckEvent(I2Cx, I2C_EVENT_MASTER_MODE_SELECT));

  // Send Address

  I2C_Send7bitAddress(I2Cx, SlaveAddress, I2C_Direction_Receiver);

  // EV6

  Timed(!I2C_GetFlagStatus(I2Cx, I2C_FLAG_ADDR));

  if (nbyte == 1) { /* read 1 byte */ ... }
  else if (nbyte == 2)  { /* read 2 bytes */ ... }
  else  { /* read 3 or more bytes */ ...  }

 // Wait for stop

  Timed(I2C_GetFlagStatus(I2Cx, I2C_FLAG_STOPF));
  return Success;

 errReturn:
  return Error;
}
```

Listing 9.3: I²C Read Transaction

```
 if (nbyte == 1) {
     // Clear Ack bit

     I2C_AcknowledgeConfig(I2Cx, DISABLE);

     // EV6_1 -- must be atomic -- Clear ADDR, generate STOP

     __disable_irq();
     (void) I2Cx->SR2;
     I2C_GenerateSTOP(I2Cx,ENABLE);
     __enable_irq();

     // Receive data   EV7

     Timed(!I2C_GetFlagStatus(I2Cx, I2C_FLAG_RXNE));
     *buf++ = I2C_ReceiveData(I2Cx);

   }
```

Listing 9.4: $I^2C$ Read 1 Byte

```
else if (nbyte == 2) {
     // Set POS flag

     I2C_NACKPositionConfig(I2Cx, I2C_NACKPosition_Next);

     // EV6_1 -- must be atomic and in this order

     __disable_irq();
     (void) I2Cx->SR2;                       // Clear ADDR flag
     I2C_AcknowledgeConfig(I2Cx, DISABLE); // Clear Ack bit
     __enable_irq();

     // EV7_3  -- Wait for BTF, program stop, read data twice

     Timed(!I2C_GetFlagStatus(I2Cx, I2C_FLAG_BTF));

     __disable_irq();
     I2C_GenerateSTOP(I2Cx,ENABLE);
     *buf++ = I2Cx->DR;
     __enable_irq();

     *buf++ = I2Cx->DR;

   }
```

Listing 9.5: $I^2C$ Read 2 Bytes

```
else {
    (void) I2Cx->SR2;    // Clear ADDR flag
    while (nbyte-- != 3)
      {
        // EV7 -- cannot guarantee 1 transfer completion time,
        //  wait for BTF instead of RXNE

        Timed(!I2C_GetFlagStatus(I2Cx, I2C_FLAG_BTF));
        *buf++ = I2C_ReceiveData(I2Cx);
      }

    Timed(!I2C_GetFlagStatus(I2Cx, I2C_FLAG_BTF));

    // EV7_2 -- Figure 1 has an error, doesn't read N-2 !

    I2C_AcknowledgeConfig(I2Cx, DISABLE); // clear ack bit

    __disable_irq();
    *buf++ = I2C_ReceiveData(I2Cx); // receive byte N-2
    I2C_GenerateSTOP(I2Cx,ENABLE);  // program stop
    __enable_irq();

    *buf++ = I2C_ReceiveData(I2Cx); // receive byte N-1

    // wait for byte N

    Timed(!I2C_CheckEvent(I2Cx, I2C_EVENT_MASTER_BYTE_RECEIVED));
    *buf++ = I2C_ReceiveData(I2Cx);

    nbyte = 0;

  }
```

Listing 9.6: I²C Read 3 or More Bytes

# Chapter 10

# Timers

Micro-controllers, such as the STM32 utilize hardware timers to generate signals of various frequencies, generate pulse-width-modulated (PWM) outputs, measure input pulses, and trigger events at known frequencies or delays. The STM32 parts have several different types of timer peripherals which vary in their configurability. The simplest timers (TIM6 and TIM7) are primarily limited to generating signals of a known frequency or pulses of fixed width. While more sophisticated timers add additional hardware to utilize such a generated frequency to independently generate signals with specific pulse widths or measure such signals. In this chapter we show how timers can be used to control the intensity of the ST7735 backlight (by modulating its enable signal) and to control common hobby servos.

An example of a basic timer is illustrated in Figure 10.1. This timer has four components – a controller, a prescaler (PSC), an "auto-reload" register (ARR) and a counter (CNT). The function of the prescaler is to divide a reference clock to lower frequency. The STM32 timers have 16-bit prescaler registers and can divide the reference clock by any value 1..65535. For example, the 24Mhz system clock of the STM32 VL Discovery could be used to generate a 1 Mhz count frequency with a prescaler of 23 (0..23 == 24 values). The counter register can be configured to count up, down, or up/down and to be reloaded from the auto reload register whenever it wraps around (an "update event") or to stop when it wraps around. The basic timer generates an output event (TGRO) which can be configured to occur on an update event or when the counter is enabled (for example on a GPIO input).

To understand the three counter modes consider Figure 10.2. In these examples, we assume a prescaler of 1 (counter clock is half the internal clock), and a auto reload value of 3. Notice that in "Up" mode, the counter increments

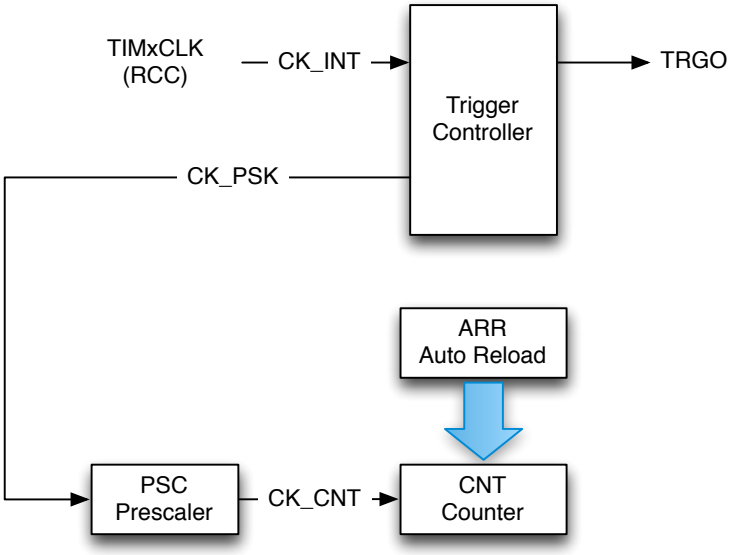



Figure 10.1: Basic Timer

from 0 to 3 (ARR) and then is reset to 0. When the reset occurs, an "update event" is generated. This update event may be tied to TRGO, or in more complex timers with capture/compare channels it may have additional effects (described below). Similarly, in "Down" mode, the counter decrements from 3 to 0 and then is reset to 3 (ARR). In Down mode, an update "event" (UEV) is generated when the counter is reset to ARR. Finally, in Up/Down mode, the counter increments to ARR, then decrements to 0, and repeats. A UEV is generated before each reversal with the effect that the period in Up/Down mode is one shorter than in either Up or Down mode.

Many timers extend this basic module with the addition of counter channels such as the one illustrated in Figure 10.3. The "x" refers to the channel number – frequently, timers support multiple channels. With this modest additional hardware, an output can be generated whenever the count register reaches a specific value or the counter register can be captured when a specific input event occurs (possibly a prescaled input clock).

An important use of counter channels is the generation of precisely timed pulses. There are two variations of this use – "one-pulse" pulses, in which a single pulse is generated, and pulse width modulation, in which a

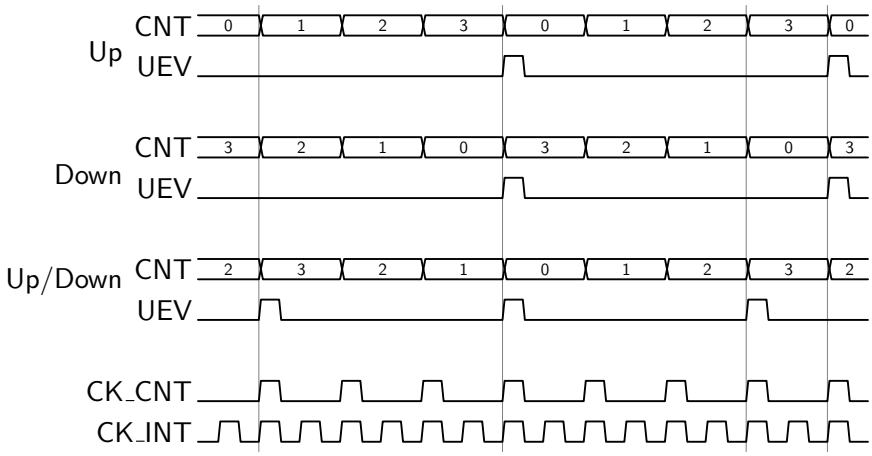



Figure 10.2: Counter Modes (ARR=3, PSC=1)

series of pulses is generated with the counter UEV period. The pulse width is controlled by the Capture/Compare Register (CCR). For example, the channel output (OCxREF) may tied to whether the CNT register is greater (or less) than the Compare register. In Figure 10.4 we illustrate the use of two channels for one-pulse and PWM outputs. Here we assume that the ARR is 7 and the CCR is 3. In PWM mode, ARR controls the period, and CCR controls the pulse width (and hence the duty cycle). In one-pulse mode, the pulse begins CCR cycles after an initial trigger event, and has a width of ARR-CRR. It is possible to use multiple channels to create a set of synchronized, pulses beginning at precise delays from each other.

A timer channel may also be used to measure pulse widths – in effect decoding pwm signals. There are many other configuration options for the STM32 timers including mechanisms to synchronize multiple timers both to each other and to external signals.

In the remainder of this chapter we consider two timer applications including PWM output (Section 10.1), input pulse measurement (Section 10.2). In Chapter 13 we show how to use a timer to control DMA transfers for an audio player and in Chapter 14 we use a timer to sample and analog input at regular intervals.

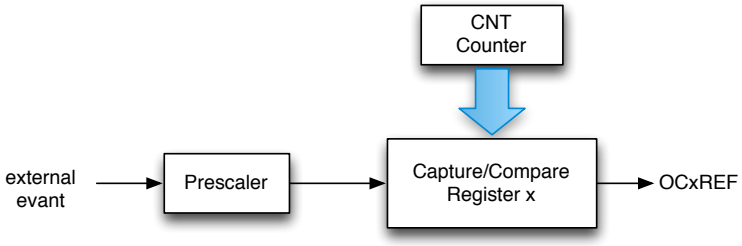



Figure 10.3: Timer Channel

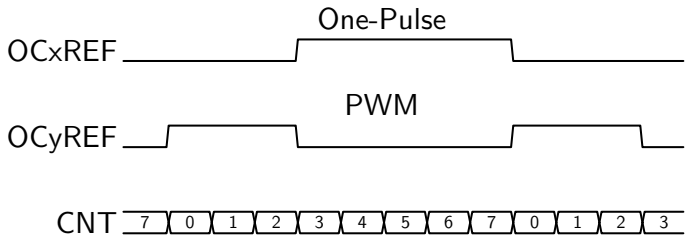



Figure 10.4: Pulse Width Modulation (PWM)

## 10.1 PWM Output

In this section we consider two examples utilizing pulse-width-modulated output – a backlight control for the 7735 LCD and a hobby servo control.

### 7735 Backlight

The 7735 backlight consists of a number of LEDs which are turned on by pulling the 7735 backlight control pin (PA1) low and off by pulling it high. It is possible to "dim" LEDs by applying a PWM signal which modulates their duty cycle. In this section, we show how to configure a timer to allow the intensity of the 7735 backlight to be modified under program control. The library code to configure timers is in `stm32f10x_tim.[ch]`.

By consulting the STM32 VL Discovery User Manual [14] we find that PA1 is "conveniently" associated with `TIM2_CH2` – that is, channel 2 of timer

TIM2 can drive the pin. [1]

```
TIM_TimeBaseInitTypeDef  TIM_TimeBaseStructure;
TIM_OCInitTypeDef  TIM_OCInitStructure;

// enable timer clock

RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2, ENABLE);

// configure timer
// PWM frequency = 100 hz with 24,000,000 hz system clock
//   24,000,000/240 = 100,000
//   100,000/1000 = 100

TIM_TimeBaseStructInit(&TIM_TimeBaseStructure);
TIM_TimeBaseStructure.TIM_Prescaler
              = SystemCoreClock/100000 - 1; // 0..239
TIM_TimeBaseStructure.TIM_Period = 1000 - 1; // 0..999
TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
TIM_TimeBaseInit(TIM2, &TIM_TimeBaseStructure);

// PWM1 Mode configuration: Channel2
//   Edge-aligned; not single pulse mode

TIM_OCStructInit(&TIM_OCInitStructure);
TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;
TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
TIM_OC2Init(TIM2, &TIM_OCInitStructure);

// Enable Timer

TIM_Cmd(TIM2, ENABLE);
```

Listing 10.1: Timer Configuration for PWM

Listing 10.1 illustrates the steps necessary to configure timer 2 to operate with a period of 100 Hz and 1000 steps of the timer clock. This allows us to define a pulse width output from 0-100% with a precision of 0.1%. The major configuration parameters are the prescaler, period, and count mode (up !). The output channel is configured in PWM (repetitive) mode (there are actually two variations – edge-aligned and center-aligned – here we choose edge-aligned. Not shown is the code to reconfigure PA1 for "alternative func-

[1] One of the hardest optimization problems when designing a micro-controller based system is to choose pins in a manner that enables access to all the necessary hardware peripherals. Careful advance planning can save a lot of grief !

tion push-pull" mode. Note, the when configuring the output channel, the "number" is embedded in the name of the call – in this case `TIM_OC2Init` initializes channel 2, similarly `TIM_OC4Init` initializes channel 4. The pulse-width `pw` (0..999) can be set with the following command. (Again, the channel number is embedded in the procedure name – `TIM_SetCompare2`.)

```
TIM_SetCompare2(TIM2, pw);
```

### Exercise 10.1 Ramping LED

Write an application that displays a single color on the 7735 LCD and repeatedly "ramps" the backlight up and down (fades in and out) at a rate of 2 ms per step (2 seconds to fade in, 2 seconds to fade out).

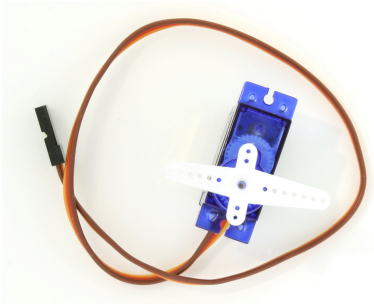


Figure 10.5: Typical Hobby Servo

### Exercise 10.2 Hobby Servo Control

The servos commonly used in radio controlled vehicles are easily controlled using the STM32 timers. A typical servo is illustrated in Figure 10.5. The servo consists of a geared motor with a moving lever arm (or wheel) and a simple electronic control used to set the lever angle. These servos typically have three connections – power (4-6 volts) (usually the middle wire), ground (brown or black), and a control signal. Servos are available in a wide range of sizes from a few grams to more than 20 grams depending upon the power requirements. However, the all work in a similar fashion.

The control protocol is very simple – every 20ms a pulse is sent to the servo. The width of the pulse determines the servo position. This is illustrated in Figure 10.6. A 1.5ms pulse corresponds to "center" (45°), 1ms corresponds

to “full left” (0°) and 2ms corresponds to “full right” (90°). Pulses between 1ms and 2ms can be used to set any angle between 0° and 90°.

Servos have a particularly simple electrical configuration – ground, power, and signal. While textemmost servos use the center of three connectors for power, it is important to check this for any servo you use. There are many references on the web. The signal wire can be connected directly to the timer output channel pin. The power signal should be 5V or less.

Configure a timer to control a pair of hobby servos using two timer channels (pins PB8 and PB9 are good choices !). Use the Nunchuk joystick to control their position. For this exercise it is OK to use USB power if the servos are quite small **and** you use a flyback diode to protect the discovery board. However, it would be better to use a separate power supply (a 3-cell battery pack would be fine). Just remember to connect the battery ground to the STM32 ground.

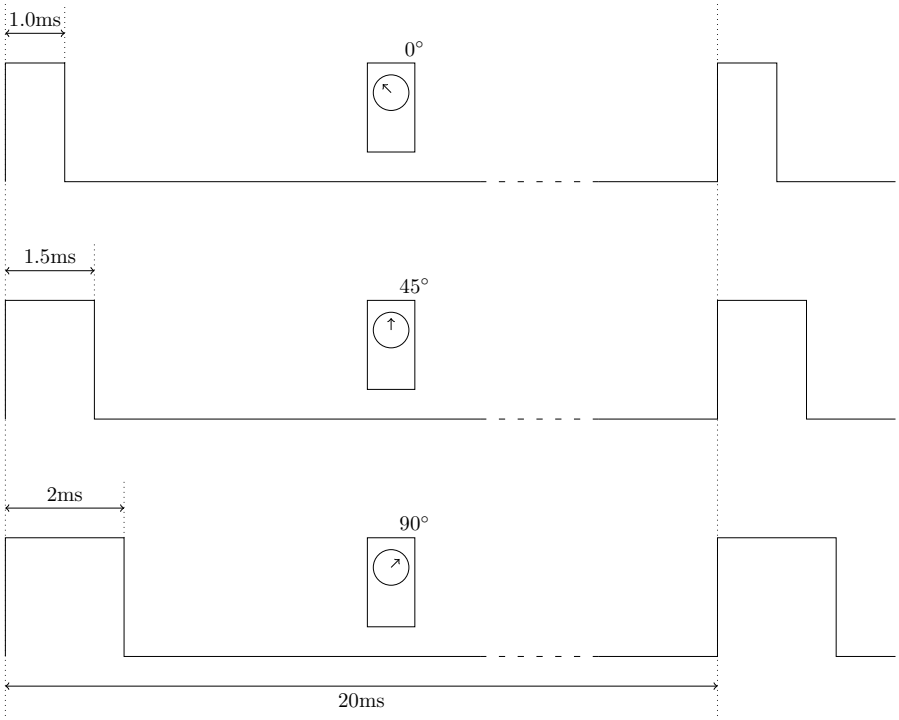



Figure 10.6: Servo Control Pulses

## 10.2 Input Capture

In generating PWM output signals we use the "compare" feature of the capture/compare register. In this section we discuss "capture". The capture registers provide a mechanism to monitor an input pin and, based upon a programmed edge, capture the current value of the corresponding timer counter. The main purpose of the capture registers is to enable measuring, relative to a time reference, when events happen. It is possible to tie multiple capture registers to a single input, capturing the times of both rising and falling edges, to measure pulse widths. It is possible to use input events to reset the timer counter, to use input values to enable a timer counter, and to synchronize multiple timers. In this section we will show how input capture can be used in conjunction with PWM output to control commonly available ultrasonic ranging devices such as that illustrated in Figure 10.7.

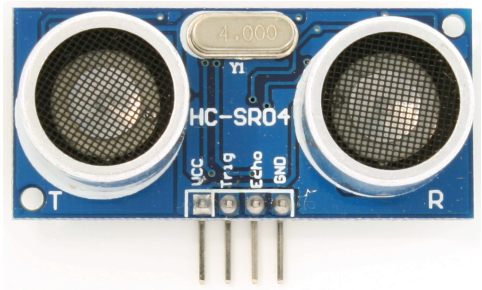



Figure 10.7: Ultrasonic Sensor

The HC-SR04 ultrasonic ranging module is capable of 3mm resolution in the range 20-500mm. It requires a 5V supply. In operation, the module is triggered by delivering it a $10\mu s$ pulse. Some time later an "echo" pulse is generated whose length is proportional to the measured distance as illustrated in Figure 10.8 and defined by the following formula where $pw$ is the echo pulse.

$$distance = pw * \frac{cm}{58\mu s}$$

If the distance is less than 20mm or greater than 500 mm, a 38ms pulse is returned. Internally, the ultrasonic controller circuit generates an 8 pulse 40 kHz signal which drives the transducer.

In the remainder of this section we will describe to use two timers – one for output and one for input to control such an ultrasonic ranger completely autonomously. Meaning that after setting up the timers, an application need

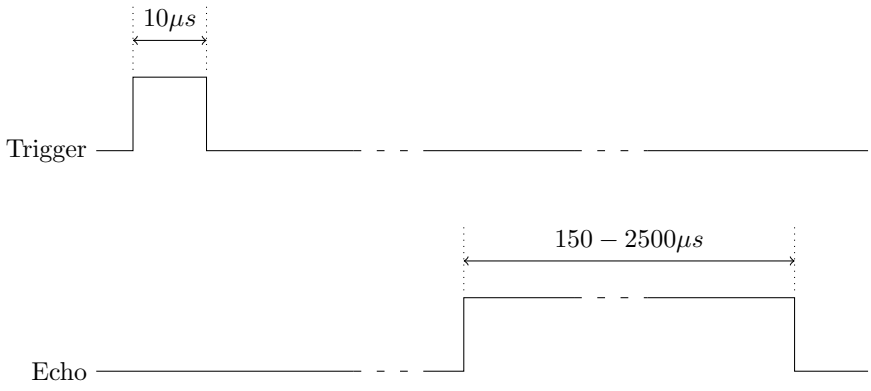



Figure 10.8: Ultrasonic Sensor Protocol

only read a capture register to learn the most recent distance measurement. With the addition of interrupts (see Chapter 11, it is possible to notify the application whenever a new measurement is available. Here we use a continuous measurement process. Note that when using multiple ranger modules, it is advisable that they not be active continuously. With a small amount of external hardware and a few GPIO pins, it is possible to multiplex the timing hardware to control an arbitrary number of ranging modules with a just two timers.

As mentioned, we use two timers – one to generate the trigger pulse and one to measure the echo pulse. We discussed how to generate a pwm output in Section 10.1 and now we leave it as an exercise to the reader to utilize timer TIM4 and Pin PB9 to generate trigger pulses at a 10Hz rate.

To measure the echo pulse, we will use timer TIM1 connected to PA8. The architecture of the various timers is quite complex. We exploit two key concepts:

1. Pairs of capture registers (1,2) and (3,4) can be "coupled" to enable capture at opposite edges of a single input.

2. Timer counters can be configured as slaves to capture inputs 1 and 2, for example, to reset the counter on a specified input event.

The details of setting up capture registers can be quite complex, in the following, we refer to the simplified model illustrated in Figure 10.9. In this figure two capture registers are illustrated along with the functional path from inputs (t1 and t2). The inputs are (optionally) filtered and signals generated on rising and falling edges. Any of the four edge signals can be selected and, after (optionally) dividing, used to trigger the capture event. Thus, capture register (channel) 1 or 2 can be loaded on rising or falling edges on either input t1 or t2. In addition two signals TI1FP1 and TI2FP2 can be used to control the timer counter, for example, causing it to be reset on the select input edge.

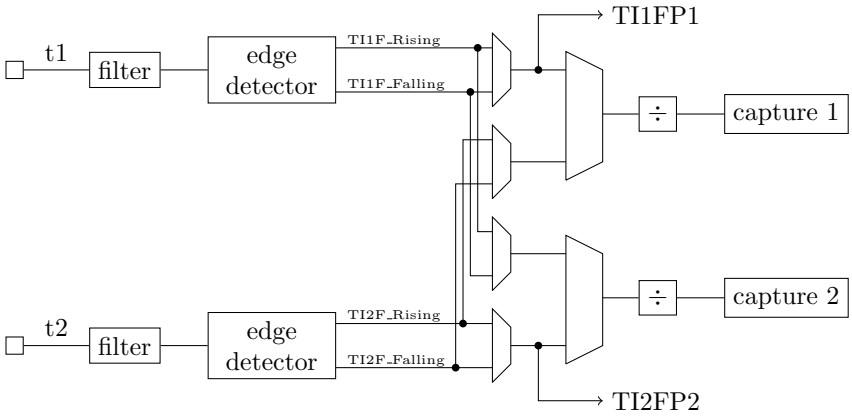



Figure 10.9: Capture Circuit

To configure a TIM1 to measure the ultrasonic echo pulse we must do the following (in addition to pin and configuration and clock distribution !):

1. Configure TIM1 prescaler and period.
2. Configure channel 1 to latch the timer on a rising input on t1.
3. Configure channel 2 to latch the timer on a falling input on t2.
4. Configure TIM1 in slave mode to reset on the capture 1 event.

Task (1) is identical to that for the trigger generation; although you may wish to use a longer period. Configuration for channel 1 follows:

```
TIM_ICInitStructure.TIM_Channel = TIM_Channel_1;
TIM_ICInitStructure.TIM_ICPolarity = TIM_ICPolarity_Rising;
```

```
TIM_ICInitStructure.TIM_ICSelection = TIM_ICSelection_DirectTI;
TIM_ICInitStructure.TIM_ICPrescaler = 0;
TIM_ICInitStructure.TIM_ICFilter = 0;
TIM_ICInit(TIM1, &TIM_ICInitStructure);
```

We require no filtering or prescaling (dividing) of the input signal. We want to capture on rising edges, using the T1 input. Configuring channel 2 has the following differences:

```
TIM_ICInitStructure.TIM_ICPolarity = TIM_ICPolarity_Falling;
TIM_ICInitStructure.TIM_ICSelection = TIM_ICSelection_IndirectTI;
```

Finally we must configure the timer slave mode with TI1FP1 as reset signal:

```
TIM_SelectInputTrigger(TIM1, TIM_TS_TI1FP1);
TIM_SelectSlaveMode(TIM1, TIM_SlaveMode_Reset);
TIM_SelectMasterSlaveMode(TIM1, TIM_MasterSlaveMode_Enable);
```

### Exercise 10.3 Ultrasonic Sensor

Write an application which tracks, by printing over the USART, the distance, in centimeters, measured by an ultrasonic sensor every 100ms.

# Chapter 11

# Interrupts

Interrupts are a fundamental hardware mechanism to enable peripherals to notify software of critical events. [1] For example, we may wish to generate an analog output signal at precise intervals in order to play an audio file. One way to achieve this is to configure a timer to generate interrupts at precise intervals. When the configured interrupt occurs, the processor switches execution from the application program to a special interrupt handler which can then "service" the interrupt event by transferring a data sample to the analog output. Once the interrupt handler has completed its task, the processor resumes execution of the application program. Interrupts are also important in communication – for example, notifying the processor when a character has arrived at a UART. In this chapter we discuss how interrupts work in the STM32 (more generally in the Cortex-M3) micro-controller family and present several concrete examples to demonstrate their use.

We've actually been using interrupts throughout this book to implement our delay function as illustrated by the code fragment in Listing 11.1. Within `main`, we configure the Cortex-M3 "SysTick" to trigger an interrupt every milli-second. Furthermore, we define an interrupt handler, `SysTick_Handler`, to be executed when the SysTick interrupt occurs. This handler decrements the variable `TimingDelay` and then returns control to the application program. The application program may wait for a precise period by calling the procedure `Delay` with an interval and then waiting for this interval to elapse (literally be counted down by the handler).

Interrupts may be triggered by many possible events including the sys-

[1] Interrupts are a special case of exceptions, which may include internal events such as access violations.

```
main(){
  ...
  if (SysTick_Config(SystemCoreClock / 1000))
     while (1);
  ...
}

static __IO uint32_t TimingDelay;

void Delay(uint32_t nTime){
  TimingDelay = nTime;
while(TimingDelay != 0);
}

void SysTick_Handler(void){
  if (TimingDelay != 0x00)
  TimingDelay--;
}
```

Listing 11.1: SysTick Interrupt

tem timer, memory access faults, external resets, and by all of the various peripherals provided on the STM32 processor. It is easiest to view each possible interrupt source as a separate signal that is monitored by the processor core during program execution. If the core detects a valid interrupt signal and is configured to accept interrupt requests, it reacts by saving the state of the currently executing program on the program stack and executing a handler (sometimes called an interrupt service routine) corresponding to the accepted interrupt. When the handler terminates, the saved program state is restored and normal program execution is resumed.

The concept of interrupts, and more generally, exceptions can be relatively difficult to grasp. Recall that programs in a language such as C are compiled into assembly code (the symbolic representation of machine instructions) from which machine code is generated. In the STM32, this machine code is copied into flash memory when programming the device (we do this through gdb) and executed when the processor is reset. The basic execution model is:

```
while (1){
  inst = *pc++;
  eval(inst);
}
```

The processor can be viewed as a machine code interpreter which reads instructions from memory and evaluates them. The program counter, `pc`, holds the address of the next instruction to execute. Consider the following 'C' statement from the SysTick handler:

```
TimingDelay--;
```

The compiler translates this statement into three assembly language steps which load (ldr) the value of TimingDelay, decrement the value (subs) and write the decremented value back (str). This listing includes the 6 bytes of machine code (0x68a1, 0x3a01, 0x601a) generated by the assembler. While outside the scope of this book, it's important to realize that assembly language is fundamentally a human readable form of the binary machine code. The process of linking assigns these instructions to fixed memory addresses.

```
  681a  ldr  r2, [r3, #0]
  3a01  subs r2, #1
  601a  str  r2, [r3, #0]
```

Implementing interrupts in the processor requires extending this model slightly:

```
while (1) {
  if (interrupt_pending()) {
    save_state();
    pc = find_handler();
  } else {
    inst = *pc++;
    eval(inst);
  }
}
```

On every "cycle" a test is made to determine if an interrupt is pending – this literally corresponds to checking if a hardware input is 1 or 0. If so, the current program state (including the program counter) is saved, an interrupt handler address is found, and execution continues with the handler. When the handler completes execution, the preempted state is restored and execution of the application continues from the point of interruption. Note for example, that interrupts occur at machine instruction boundaries and not a "C" statement boundaries. As we shall see, understanding this is fundamental to writing reliable interrupt code.

In the case of the SysTick notice that the handler only modifies the shared data (`TimingDelay`) when it is non-zero and the application only modifies the shared data when it is zero. Guarantees like this are important because

interrupt handlers cause application code to be suspended at unpredictable locations and therefore the data structures that are used to communicate between handlers and application code must be very carefully constructed to ensure that they work correctly. Consider the chaos that might result if an application program and interrupt handler both access a linked list – interrupt handlers had better not access links that are in the process of being moved by the application code. This kind of conflict is called a "race condition." We present an example of using an interrupt handler to service a UART where the application and handler share two data queues – the code is carefully crafted to avoid potential data races.

The SysTick example demonstrates some typical interactions between interrupt handlers and applications. The handler executes very briefly – just a few machine instructions – in order to update some information shared with the application program. To achieve reliability interrupt handlers must satisfy three properties:

1. They must execute briefly.
2. Their execution time must be predictable (e.g. they must never wait)
3. Their use of "shared data" must be carefully managed.

This model is of course an over-simplification. However, for the subsequent discussion of interrupts, it is a sufficient model. Key questions that we will address are – how do we enable peripherals to generate pending interrupts, how do we ensure that our handlers are executed, and how do we write reliable handlers ? Not illustrated in this model is the concept of interrupt priority – when more than two interrupts are requested, how is the selection made. Neither is interrupt preemption addressed – if an interrupt handler is executing, is it possible for other handlers to be called recursively ?

The remainder of this chapter is organized as follows. We begin with a discussion of the Cortex-M3 interrupt (exception) model including the various exception types, stacks, and privileges. We then discuss the interrupt vector table – the mechanism by which the Cortex-M3 core associates handlers with specific interrupt causes, and the Nested Vector Interrupt Controller (NVIC) which the Cortex-M3 uses to select among competing interrupt requests.

## 11.1 Cortex-M3 Exception Model

The Cortex-M3 processors support two modes of operation, Thread mode and Handler mode. Thread mode is entered on Reset, and can be entered through an exception return. Handler mode is entered as a result of an exception being taken. The Cortex-M3 processors support two distinct stack pointers – as illustrated in Figure 11.1. Out of reset, all code utilizes the main stack; however, the processor can be configured so that application code utilizes a separate "process stack." Whenever the processor invokes an exception, the handler executes using the main stack, and when execution returns from an exception, the processor resumes using the stack used prior to the exception. When invoking an exception, the state of the currently executing code must be saved on the main stack to be restored when the exception handler terminates. With single threaded applications, such as the examples we have considered thus far, there is no significant advantage to utilizing separate stacks; however, when using an operating system supporting threads, it is generally considered desirable to use a separate stack for exception handling and execution within the OS kernel.

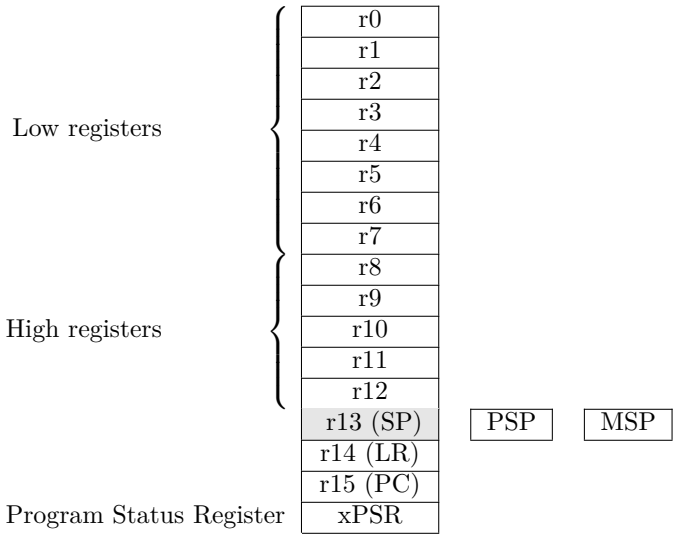



Figure 11.1: Processor Register Set

State saving and restoring is a collaboration between the processor hardware and the exception handlers. The Cortex-M3 architecture assumes that the exception handler code will obey the Arm Architecture Procedure Call

Standard [2] which dictates that all procedures (exception handlers included) save and restore specific registers if they are modified. The Cortex-M3 exception hardware takes responsibility for saving any other registers. Specifically, when an exception is taken, the processor pushes eight registers – xPSR, PC, LR, r12, r3, r2, r1, and r0 – onto the main stack as illustrated in Figure 11.1. When returning from an exception handler, the processor automatically pops these off the main stack. Upon entry to the exception handler, the link register (LR) contains a special value which controls the exception return.

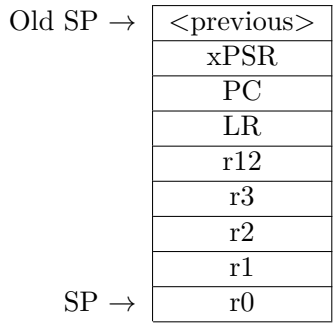



Figure 11.2: Main Stack Contents After Preemption

The Cortex-M3 processor supports multiple exception priorities. Even if an exception handler is currently executing, a higher-priority exception can be invoked preempting the current handler (priorities are defined as integers with smaller integers having higher priorities). It this case, the state of the preempted handler will be pushed on the main stack prior to executing the new handler. The highest priority exception is Reset (-3), which can preempt all others. The priority of most exceptions can be set under program control.

The Cortex-M3 processor core locates the interrupt (exception) handlers through a vector table. Each entry in the table consists of the address of an interrupt handler. The table is indexed by the unique number of each interrupt source. The format of the STM32 vector table is defined both by the Cortex-M3 reference manual [1] and by the appropriate STM32 reference manual [21, 20]. A fragment of this table for the STM32 F100xx devices is illustrated in Table 11.1.

The first 16 entries (through SysTick_Handler) are defined by the CortexM3 specification and the remainder are processor specific. The first entry in the table is not the address of an interrupt handler, but rather the address of the initial stack. At reset, the core loads the stack pointer from memory location 0x0000_0000 and begins execution at the the Reset_Handler loca-

| Name | Description | Address |
|---|---|---|
| - | Initial Stack Pointer | 0x0000_0000 |
| Reset_Handler | Executed at reset | 0x0000_0004 |
| NMI_Handler | Non-maskable interrupt | 0x0000_0008 |
| | ... | |
| SysTick_Handler | System Tick Timer | 0x0000_0010 |
| | ... | |
| EXTI0_ISRHandler | External interrupt line 0 | 0x0000_0058 |
| | ... | |
| TIM3_ISRHandler | Timer 3 interrupt | 0x0000_00B4 |
| | ... | |
| USART1_ISRHandler | USART1 global interrupt | 0x0000_00D4 |

Table 11.1: STM32 F100xx Interrupt Vector Table

tion stored at memory location 0x0000_0004. This fragment also includes the entries for TIM3, USART1 and EXTI0, which we will be considering in this chapter.

The Cortex-M3 specifies that the vector table is located beginning at memory location 0x0000_0000. There are a couple of important exceptions. The STM32, when booting from flash, "aliases" the flash memory, which starts at location 0x0800_0000 to 0x0000_0000 – thus memory reads from location 0x0000_0000 actually return the value stored at 0x0800_0000 (this and other aliases are described in in Section 2.4 of [20]). It is also possible to move the location of the active vector table at runtime (see [19]) – an important feature for supporting fixed bootloaders which may place application vector tables in other locations. The interrupt vector table is defined in the startup code and its location defined by the linker script both of which are described in Chapter 3.

The startup code discussed in Chapter 3 is designed to simplify modification of the vector table. Every interrupt vector is provided with an initial "weak" definition. To override this definition, it is sufficient to define a procedure with the correct name. For example, we previously defined a system timer handler:

```
void SysTick_Handler(void){
  if (TimingDelay != 0x00)
    TimingDelay--;
}
```

```
Reset_Handler                   DMA1_Channel6_IRQHandler
NMI_Handler                     DMA1_Channel7_IRQHandler
HardFault_Handler               ADC1_IRQHandler
MemManage_Handler               EXTI9_5_IRQHandler
BusFault_Handler                TIM1_BRK_TIM15_IRQHandler
UsageFault_Handler              TIM1_UP_TIM16_IRQHandler
SVC_Handler                     TIM1_TRG_COM_TIM17_IRQHandler
DebugMon_Handler                TIM1_CC_IRQHandler
PendSV_Handler                  TIM2_IRQHandler
SysTick_Handler                 TIM3_IRQHandler
WWDG_IRQHandler                 TIM4_IRQHandler
PVD_IRQHandler                  I2C1_EV_IRQHandler
TAMPER_IRQHandler               I2C1_ER_IRQHandler
RTC_IRQHandler                  I2C2_EV_IRQHandler
FLASH_IRQHandler                I2C2_ER_IRQHandler
RCC_IRQHandler                  SPI1_IRQHandler
EXTI0_IRQHandler                SPI2_IRQHandler
EXTI1_IRQHandler                USART1_IRQHandler
EXTI2_IRQHandler                USART2_IRQHandler
EXTI3_IRQHandler                USART3_IRQHandler
EXTI4_IRQHandler                EXTI15_10_IRQHandler
DMA1_Channel1_IRQHandler        RTCAlarm_IRQHandler
DMA1_Channel2_IRQHandler        CEC_IRQHandler
DMA1_Channel3_IRQHandler        TIM6_DAC_IRQHandler
DMA1_Channel4_IRQHandler        TIM7_IRQHandler
DMA1_Channel5_IRQHandler
```

Figure 11.3: Vector Names Defined for the Medium Density Value Line Parts

Similarly, we will define a handler for USART1 as

```
void USART1_IRQHandler(void) {
   // Check interrupt cause
...
   // Clear interrupt cause
}
```

Notice, that in contrast with the SysTick handler, most handlers must, at the very least, determine the cause of the interrupt – with the USART this might be an empty transmit buffer or a full receive buffer – and clear the interrupt cause. Determining the cause is generally performed by reading a peripheral specific status register. Clearing the interrupt is accomplished by performing a necessary action (e.g. reading a data register) or directly resetting the corresponding status bit.

The required handler names defined in the startup code are shown in Figure 11.1. Note that different members of the STM32 family support different peripherals and hence have varying sets of handler names. Furthermore, these names are defined in the startup code and not by library functions. If in doubt, you must look at the startup sources ! Also, be careful of typos in vector names which may be challenging to diagnose.

## 11.2 Enabling Interrupts and Setting Their Priority

The Cortex-M3 core defines a sophisticated priority mechanism that allows interrupt sources to be assigned both a priority and a sub-priority. At a given priority level, two interrupt sources are serviced in order of their sub-priority (lower number takes precedence). If an interrupt handler is active and another interrupt arrives with a lower priority number, the the active handler will be preempted. The Cortex-M3 defines up to 8 priority level bits that may be split among the priority and sub-priority fields. The STM32 processor implements only 4 of these bits. Throughout this book we utilize a configuration where 0 bits are allocated to priority and 4 bits are allocated to sub-priority. In other words, we choose not to enable interrupt preemption.

The interrupt priority mechanism is managed through the NVIC (Nested Vectored Interrupt Controller) that is a standard peripheral for all Cortex-M3 based processors. The NVIC provides the following functions for every interrupt source:

- Priority and sub-priority configuration.
- Enable (disable) interrupt.
- Set/Clear interrupt pending bit.

## 11.3 NVIC Configuration

The STM32 NVIC supports 16 distinct priority and sub-priority levels supported by 4 bits which are partitioned between these two functions. Interrupts with different priorities can preempt each other (lower number priorities have precedence) while sub-priorities within a single priority only affect the choice of interrupt taken when two or more are pending. For the examples in

this chapter we use 0 bits for priority. In Chapter 16 we will discuss FreeRTOS, a real-time operating system, that requires using 4 bits for priority.

```
#include <misc.h>

/*
 * NVIC_PriorityGroup_0  0 bits priority, 4 bits subgroup
 * NVIC_PriorityGroup_1  1 bits priority, 3 bits subgroup
 * NVIC_PriorityGroup_2  2 bits priority, 2 bits subgroup
 * NVIC_PriorityGroup_3  3 bits priority, 1 bits subgroup
 * NVIC_PriorityGroup_4  4 bits priority, 0 bits subgroup
*/

NVIC_PriorityGroupConfig(NVIC_PriorityGroup_0);
```

The following code fragment configures and enables the TIM2 interrupt. Note that this should follow any device configuration !!!

```
NVIC_InitTypeDef NVIC_InitStructure;

// No  StructInit call in API

NVIC_InitStructure.NVIC_IRQChannel = TIM2_IRQn;
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 3;
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
NVIC_Init(&NVIC_InitStructure);
```

The IRQChannel number may be device specific. These are defined in `stm32f10x.h` which you may consult for the correct constant name – do not copy the constant value into your code ! Notice that this header file depends upon device specific definitions – in our make file we define `STM32F10X_MD_VL` in our compilation flags as discussed in Chapter 3.

## 11.4 Example: Timer Interrupts

In Section 10.1 we showed how to configure timer TIM2 to control the LCD back light. In this example, we show how to enable the TIM2 interrupt. Since this builds upon work you have seen before, we present a basic outline in Listing 11.2. In addition to configuring the timer, it is necessary to configure the NVIC for the appropriate interrupt vector and to enable the timer to generate interrupts. Timers can generate interrupts on multiple conditions – here we choose to trigger interrupts whenever the counter is updated. Finally, we need a handler which, at a minimum, clears the pending interrupt.

### Exercise 11.1 Timer Interrupt – Blinking LED

Complete the timer interrupt program so that it blinks the green LED (PA9) at a 1hz rate (half-second on/ half-second off).

```
  // Configure clocks for GPIOA and TIM2
  ...
  // Configure NVIC -- see preceding section
  ...
 // Configure Timer
  ...
 // Enable Timer Interrupt, enable timer

  TIM_ITConfig(TIM2, TIM_IT_Update, ENABLE);
  TIM_Cmd(TIM2, ENABLE);

  while(1) {   /* do nothing */ }
}

void TIM2_IRQHandler(void)
{
  /* do something */
  TIM_ClearITPendingBit(TIM2,TIM_IT_Update);
}
```

Listing 11.2: Timer Tick Interrupt

## 11.5 Example: Interrupt Driven Serial Communications

The fundamental weakness of the serial communication code presented in Chapter 9 is that unless the user code is constantly polling the USART and is prepared to receive characters as soon as they arrive, there is a high probability that the (single) receive buffer will overflow and characters will be lost. Ideally, we want a solution that does not require such a high degree of attention from the user code and which also can guarantee that characters are not lost. Similarly, when the user code wishes to transmit a string it must wait for the transmit buffer to empty after each character sent.

A partial solution is to create larger transmit and receive buffers in software and utilize an interrupt handler to manage the details of receiving and transmitting characters. The interrupt handler is a user supplied routine that is executed, asynchronously to the user code, whenever the USART transmit

buffer becomes empty or the USART receive buffer becomes full. This relieves the user program of the burden of monitoring the USART buffers and, by providing additional buffer space, helps decouple the user program from the communication process. It is only a partial solution because once the larger software receive buffer is full, any additional characters received will be lost. A complete solution, described in Section 11.5, utilizes additional USART flow control signals to block communication when no future space exists.

As discussed in Chapter 5, polling based implementations of `getchar` and `putchar` suffer significant performance issues. In the case of `putchar`, the application code is slowed to the USART baud rate whenever it attempts to transmit back-to-back characters. The situation for `getchar` is even more dire – if the application code doesn't constantly monitor the USART receive data register there is a significant risk of lost data. In this section we show how interrupts in conjunction with software buffers can be used to alleviate, but not eliminate, the problem. A complete solution, as discussed in Section 11.5 requires the use of hardware flow control in conjunction with interrupts.

In an interrupt-driven USART responsibility for reception and transmission of data is split between the application code and interrupt code. The interrupt code is called whenever configured events occur; for example, when the transmit data register is empty or the receive data register is full. The interrupt code is responsible for removing the interrupt condition. In the case of a full receive data register, the interrupt handler removes the interrupt condition by reading the received data. The application code and interrupt handler communicate through a pair of software buffers implemented as queues as illustrated in Figure 11.4.

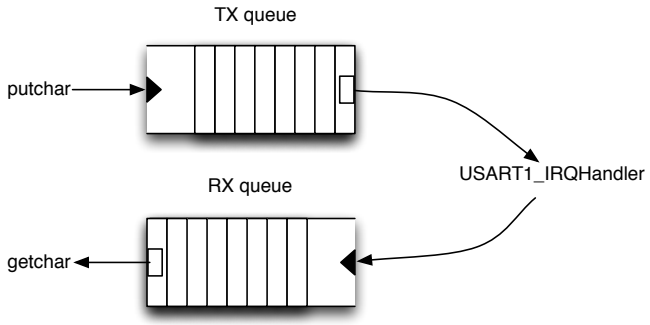



Figure 11.4: Interrupt Driven USART Software

The basic idea is simple. Whenever the application code executes `putchar`, a character is added to the TX queue and whenever the application code executes `getchar`, a character is removed from the RX queue. Similarly, whenever the interrupt handler, `USART1_IRQHandler`, is called, the handler removes the interrupting condition by moving a character from the TX queue to the transmit data register or by moving a character from the receive data register to the RX queue. All of the implementation difficulties arise from handling the edge cases where the two queues are either empty or full.

The design decisions for the edge cases differ for the application and interrupt code. The most important requirement for interrupt code is that it may never block. For example, if the receive data register is full and the RX queue is also full, the only way to remove the interrupt condition is to read the receive data register and discard the data. Thus, an interrupt driven USART cannot completely eliminate the lost data problem of the polling based solution – that will come with the addition of flow control. In contrast, the application code may block. For example, if the application executes `putchar` and the TX queue is full, then it may "poll" to wait for the full condition to be removed (by the interrupt handler). In this case, the application code is again slowed to the transmit rate, but only after the TX queue is filled. An important implementation decision is how large the queues should be to prevent application stalling.

Let us assume a queue data structure with two operations

```
struct Queue;
struct Queue UART1_TXq, UART1_RXq;

int Enqueue(struct Queue *q, uint8_t data);
int Dequeue(struct Queue *q, uint8_t *data);
```

The `Enqueue` operation takes as parameters a queue and a data byte; its function is to add the data to the queue. If the queue is full then Enqueue should return an error (0). Thus, reliable use of the `Enqueue` operation may require repeated calls. The `Dequeue` operation is similar but the operation removes a data byte from the queue. As with `Enqueue`, `Dequeue` returns an indicator of success and reliable operation may require multiple calls.

We further assume that the necessary interrupt hardware is enabled for two conditions – the receive data register is not empty and the transmit data register is empty. The interrupt handler code must then cope with these two possible conditions:

```
static int TxPrimed = 0;
int RxOverflow = 0;
void USART1_IRQHandler(void)
{
  if(USART_GetITStatus(USART1, USART_IT_RXNE) != RESET)
    {
      uint8_t  data;

      // buffer the data (or toss it if there's no room
      // Flow control will prevent this

      data = USART_ReceiveData(USART1) & 0xff;
      if (!Enqueue(&UART1_RXq, data))
        RxOverflow = 1;
    }

  if(USART_GetITStatus(USART1, USART_IT_TXE) != RESET)
    {
      uint8_t data;

      /* Write one byte to the transmit data register */

      if (Dequeue(&UART1_TXq, &data)){
          USART_SendData(USART1, data);
      } else {
          // if we have nothing to send, disable the interrupt
          // and wait for a kick

          USART_ITConfig(USART1, USART_IT_TXE, DISABLE);
          TxPrimed = 0;
      }
    }
}
```

Notice that the receive and transmit code both contain the essential elements of the polling implementations of `putchar` and `getchar`, they handle corner cases differently. If there is no room in the RX queue, the interrupt handler receives, but discards any data in the receive register. Whenever the interrupt handler discards data, it sets a global variable, `RxOverflow` to 1; it is up to the application code to monitor this variable and decide how to handle lost data. If the TX queue is empty, the interrupt handler cannot resolve the interrupt condition (since it has nothing to write to the transmit data register), so it disables the `USART_IT_TXE` condition. The variable `TxPrimed` is used to communicate to the application (specifically `putchar`) that the interrupt needs to be re-enabled when data is added to the TX queue.

The new implementation for `getchar` is quite simple (this can be generalized to replace the `uart_getc` implementation you developed in Exercise 5.2).

```
int getchar(void)
{
  uint8_t data;
  while (!Dequeue(&UART1_RXq, &data));
  return data;
}
```

The new implementation for `putchar` requires an extra test to determine if re-enabling the transmit interrupt condition is necessary:

```
int putchar(int c)
{
   while (!Enqueue(&UART1_TXq, c));
   if (!TxPrimed) {
     TxPrimed = 1;
     USART_ITConfig(USART1, USART_IT_TXE, ENABLE);
   }
}
```

The asynchronous interaction between interrupt handler code and application code can be quite subtle. Notice that we set `TxPrimed` before re-enabling the interrupt. If the order were reversed it would be possible for the newly enabled interrupt handler to empty the transmit queue and clear `TxPrimed` before the application set `TxPrimed`, thus losing the signal from the handler to the application code. Admittedly, the USART is sufficiently slow that this scenario is unlikely, but with interrupt code it pays to program defensively.

There remain two tasks – implementing the required queue data structure and enabling interrupts.

### Interrupt-Safe Queues

The final piece of our interrupt driven USART code is a queue implementation. The most common approach is to provide a fixed sized circular buffer with separate read and write pointers. This is illustrated in Figure 11.5.

There are several key ideas behind a circular buffer. The read pointer "chases" the write pointer around the buffer; thus, the increment functions must wrap around. Furthermore, the write pointer always references an

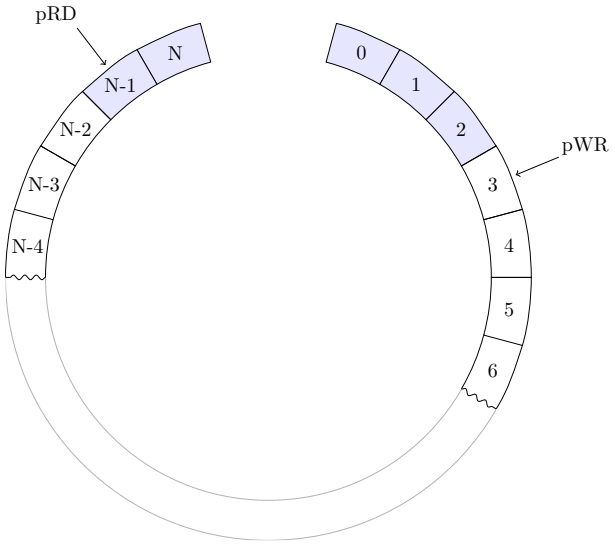



Figure 11.5: Circular Buffer

"empty" location. Thus a buffer with N+1 locations can hold at most N data elements. Thus we define the basic data structure as follows:

```
struct Queue {
  uint16_t pRD, pWR;
  uint8_t  q[QUEUE_SIZE];
};

static int QueueFull(struct Queue *q)
{
  return (((q->pWR + 1) % QUEUE_SIZE) == q->pRD);
}

static int QueueEmpty(struct Queue *q)
{
  return (q->pWR == q->pRD);
}

static int Enqueue(struct Queue *q, uint8_t data)
{
  if (QueueFull(q))
     return 0;
```

```
  else {
      q->q[q->pWR] = data;
      q->pWR = ((q->pWR + 1) ==  QUEUE_SIZE) ? 0 : q->pWR + 1;
    }
  return 1;
}

static int Dequeue(struct Queue *q, uint8_t *data)
{
  if (QueueEmpty(q))
    return 0;
  else  {
      *data = q->q[q->pRD];
      q->pRD = ((q->pRD + 1) ==  QUEUE_SIZE) ? 0 : q->pRD + 1;
  }
  return 1;
}
```

In spite of its simplicity, a shared data structure such as this offers ample opportunity for data race conditions. Notice that the `Enqueue` (`Dequeue`) operation writes (reads) the referenced queue location before updating the `qWR` (`qRD`) pointer. The order of operations is important ! As presented, the implementation assumes a single reader and a single writer. In a multi-reader or multi-writer situation (e.g. with multiple threads) a lock variable is required to prevent data races. In such a situation, the interrupt handler may not utilize a lock and hence must have exclusive access to one end of the queue.

### Hardware Flow Control

While adding interrupt driven buffers improves both the performance and reliability of data transmission, it is not sufficient to prevent buffer overrun and hence loss of data. In this section we discuss the use of two additional signals RTS (request to send) and CTS (clear to send) which may be used to enable "hardware flow" control. These signals are present on both the usb-uart bridge (assuming you are using one that exposes them) and the STM32. In practice nRTS (nCTS) from the STM32 is connected to nCTS (nRTS) of the usb-usrt bridge. [2] When the STM32 is prepared to receive data it activates (makes zero) nRTS and when it wants to stop receiving data it raises nRTS. Similarly, the usb-uart bridge will use its nRTS (connected to the STM32

[2] We use nRTS to indicate that the signal RTS is "active low".

nCTS) signal to indicate its readiness to receive data. These two pins are described in the STM32 documentation [20] as follows:

> If the RTS flow control is enabled .., then nRTS is asserted (tied low) as long as the USART receiver is ready to receive new data. When the receive register is full, nRTS is de-asserted, indicating that the transmission is expected to stop at the end of the current frame ...
>
> If the CTS flow control is enabled .., then the transmitter checks the nCTS input before transmitting the next frame. If nCTS is asserted (tied low), then the next data is transmitted (assuming that a data is to be transmitted ..), else the transmission does not occur. When nCTS is de-asserted during a transmission, the current transmission is completed before the transmitter stops.

This approach is called hardware flow control because the signaling is performed in hardware as opposed to software mechanisms that rely upon inserting special control characters into the data stream. The solution we present is not entirely hardware driven – we drive the nRTS pin through software (e.g. the interrupt handler) to indicate to the remote device that it should stop transmitting and let the USART hardware stop the transmitter whenever the remote device de-asserts nCTS.

Unfortunately, the fully automated approach is doomed to failure. To understand, consider this statement from the FTDI faq (FTDI is a prominent producer of usb-uart bridge chips:

> If [nCTS] is logic 1 it is indicating the external device cannot accept more data. the FTxxx will stop transmitting within 0-3 characters, depending on what is in the buffer.
>
> This potential 3 character overrun does occasionally present problems. Customers should be made aware the FTxxx is a USB device and not a “normal” RS232 device as seen on a PC. As such the device operates on a packet basis as opposed to a byte basis.

Thus, the behavior of real devices does not always conform to the expectations of the STM32. The only viable solution is for software to de-assert nRTS while it is still capable of receiving additional input data !. The problem is fundamentally unsolvable as there does not appear to be a clear specification that defines the amount of allowed character overrun.

The actual changes required to the code presented in Section 11.5 are modest. We must configure two additional pins for nCTS and nRTS, modify the initialization of the USART to allow nCTS to halt the transmitter, and modify both the interrupt handler and `getchar` routines. Our strategy is to define a "high water" mark in the input buffer below which the software asserts nRTS and above which it de-asserts nRTS. The high water mark provides room for additional characters to arrive after nRTS is de-asserted.

Table 5.2 provides the pin definitions for USART1. In addition to `tx` and `rx` which we previously configured we must configure `nRTS` (PA12) and `nCTS` (PA11):

```
// Configure CTS pin

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_11;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
GPIO_Init(GPIOA, &GPIO_InitStructure);

// Configure RTS pin -- software controlled

GPIO_WriteBit(GPIOA, GPIO_Pin_12, 1);          // nRTS disabled
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
GPIO_Init(GPIOA, &GPIO_InitStructure);
```

We modify slightly the USART configuration:

```
USART_InitStructure.USART_HardwareFlowControl =
USART_HardwareFlowControl_CTS;
```

and enable nRTS

```
// nRTS enabled

GPIO_WriteBit(GPIOA, GPIO_Pin_12, 0);
```

As discussed above we add code to the `getchar` routine

```
char getchar (void)
{
  uint8_t data;
  while (Dequeue(&UART1_RXq, &data, 1) != 1);

  // If the queue has fallen below high water mark, enable nRTS

  if (QueueAvail(&UART1_RXq) <= HIGH_WATER)
  GPIO_WriteBit(GPIOA, GPIO_Pin_12, 0);

  return data;
}
```

and the receive portion of the interrupt handler:

```
  if(USART_GetITStatus(USART1, USART_IT_RXNE) != RESET)
    {
      uint8_t  data;

      // clear the interrupt

      USART_ClearITPendingBit(USART1, USART_IT_RXNE);

      // buffer the data (or toss it if there's no room
      // Flow control is supposed to prevent this

      data = USART_ReceiveData(USART1) & 0xff;
      if (!Enqueue(&UART1_RXq, &data, 1))
        RxOverflow = 1;

      // If queue is above high water mark, disable nRTS

      if (QueueAvail(&UART1_RXq) > HIGH_WATER)
        GPIO_WriteBit(GPIOA, GPIO_Pin_12, 1);
    }
```

Finally, we define the high water mark in `uart.h`

```
#define HIGH_WATER (QUEUE_SIZE - 6)
```

### Exercise 11.2 Interrupt Driven Serial Communciations

- Complete the implementation of an interrupt driven UART with flow control.

- Write a simple application that reads characters and echoes them.

  You should test this by piping a large file through the serialT, capturing the results and diffing the input and output files.

- Rewrite your echo program so that it reads/writes entire lines (up to the limit of an 80-character buffer)

- Try the same tests with your polling based uart.

By now your Library of modules should look like:

```
Library
├── ff/
├── i2c.c
├── i2c.h
├── lcd.c
├── lcd.h
├── mmcbb.c
├── spi.c
├── spi.c
├── uart.c
├── uart.h
├── uart-fc.c
├── xprintf.c
└── xprintf.h
```

## 11.6 External Interrupts

The STM32 F1xx micro-controller provide up to 20 possible EXTI (external interrupt) sources; although in many cases the various sources share a single interrupt vector. The possible sources and their corresponding event/vector (for the STM32 F100) are:

| Event | Source | Vector |
|---|---|---|
| EXT0 | PA0-PG0 | EXT0_IRQHandler |
| EXT1 | PA1-PG1 | EXT1_IRQHandler |
| EXT2 | PA2-PG2 | EXT2_IRQHandler |
| EXT3 | PA3-PG3 | EXT3_IRQHandler |
| EXT4 | PA4-PG4 | EXT4_IRQHandler |
| EXT5 | PA5-PG5 | EXT9_5_IRQHandler |
| ... | ... | ... |
| EXT15 | PA15-PG15 | EXT1_10_IRQHander |
| EXT16 | PVD | PVD_IRQHandler |
| EXT17 | RTC Alarm | RTC_WKUP |
| EXT18 | USB Wakeup | not on STM32 F100 |
| EXT19 | Ethernet Wakeup | not on STM32 F100 |

Notice that only one of PAx-PGx, where x is one of 1-15, can be configured for as an EXTI source at any moment. In the event that multiple EXTI sources share a handler, pending interrupts can be determined from reading the pending register EXTI_PR. Also, any EXTI source can be "triggered" through software by setting the appropriate bit in the "software interrupt event register" EXTI_SWIER.

Configuring an external interrupt consists of several steps:

1. Configure the NVIC for the corresponding vector.
2. For GPIO pins, configure the appropriate AFIO_EXTICRx register to select the correct pin (e.g. PA0).
3. Set the trigger condition (falling/rising or both).
4. Set the appropriate bit in the event mask register.

This can all be accomplished using the standard peripheral library modules stm32f10x_nvic.[ch] and stm32f10x_exti.[ch]. For example, to configure PA0 (connected to the "user push button" on the discovery board) to trigger an interrupt on a rising edge:

```
// Connect EXTI0 to PA0

GPIO_EXTILineConfig(GPIO_PortSourceGPIOA, GPIO_PinSource0);

// Configure EXTI0 line // see stm32f10x_exti.h

EXTI_InitStructure.EXTI_Line = EXTI_Line0;
```

```
  EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
  EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising;
  EXTI_InitStructure.EXTI_LineCmd = ENABLE;
  EXTI_Init(&EXTI_InitStructure);

  // Configure NVIC EXTI0_IRQn ...
```

A basic handler should check the interrupt status and clear any pending bit;

```
void EXTI0_IRQHandler(void){ if (EXTI_GetITStatus(EXTI_Line0) !=
  RESET){ ...  EXTI_ClearITPendingBit(EXTI_Line0); } }
```

EXTI sources can also be configured in "event mode". In this mode, they do not generate an interrupt, but rather, generate an internal event that can be used to wake the processor from sleep. For example, the WFE instruction can be used by software to enter Sleep mode which may be exited by an event on any EXTI line in event mode.

### Exercise 11.3 External Interrupt

Using the code fragments presented, write a simple program that responds to rising edges (button release) events on PA0 by toggling a led3.

In writing this application, you will probably notice that depending upon how you release the button, it will often not behave as expected.

The following quotation from the reference manual is a clue to understanding what is going on:

> Note: The external wakeup lines are edge triggered, no glitches must be generated on these lines. If a falling edge on external interrupt line occurs during writing of EXTI_FTSR register, the pending bit will not be set.

Buttons and switches are notorious for "bounce" – when a button is pressed, the contacts do not separate cleanly leading to multiple spikes. This can be seen in Figure 11.6 which illustrates the actual behavior as captured on an oscilloscope – the top trace shows the voltage at the PA0 when the button is pressed and the lower trace when the button is released.

The solution is to add a de-bouncing circuit which serves as a "low-pass" filter to remove fast changing signals (glitches). A simple de-bounce circuit is illustrated in Figure 11.6 consisting of a resistor and capacitor. Because

the Discovery board has a pull-down resistor connected to the button, the suggested circuit is sub-optimal, but will eliminate the glitches as illustrated in Figure 11.6.

Try your code with the de-bounce circuit added to PA0.

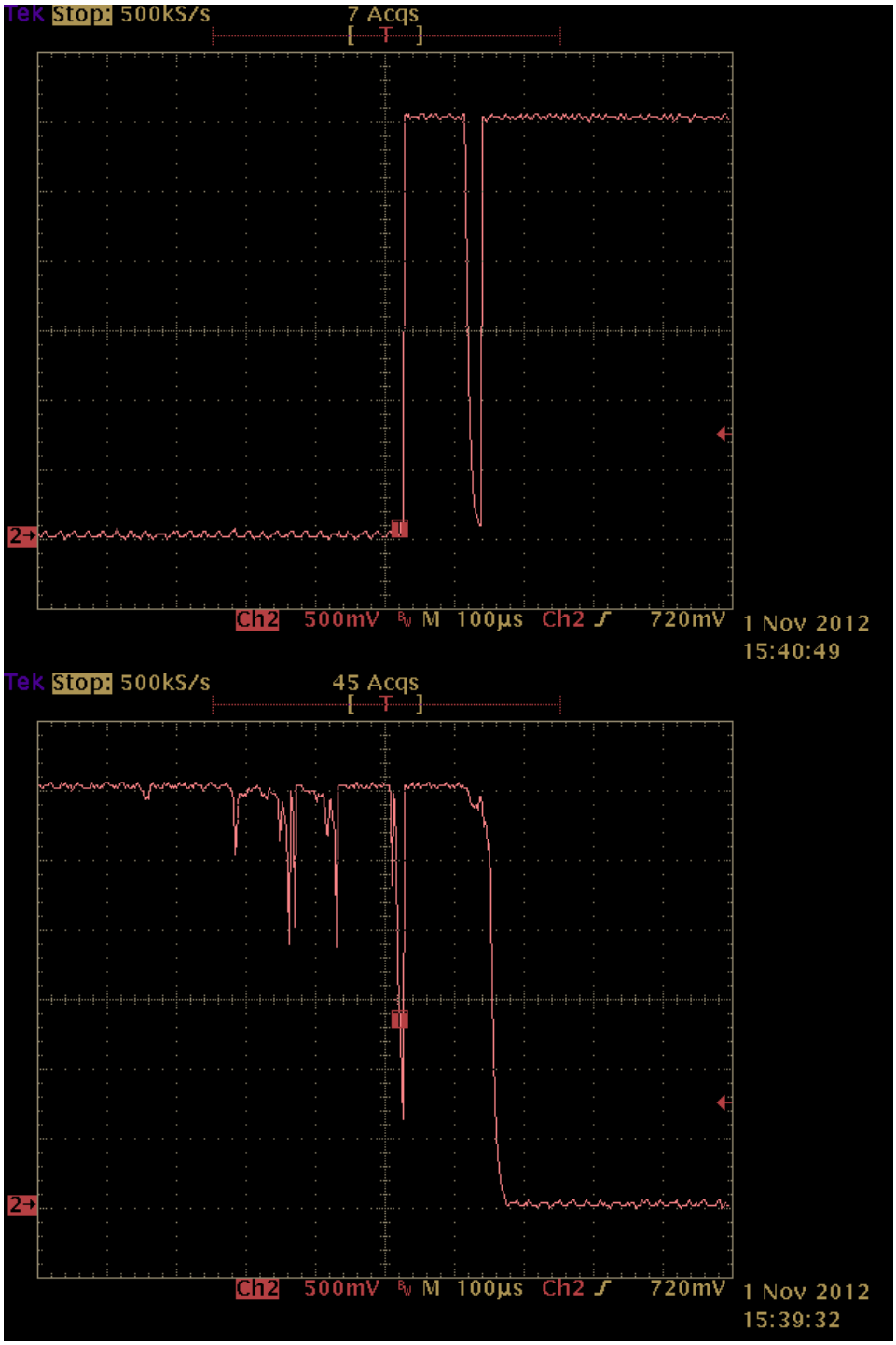





Figure 11.6: Button press/release

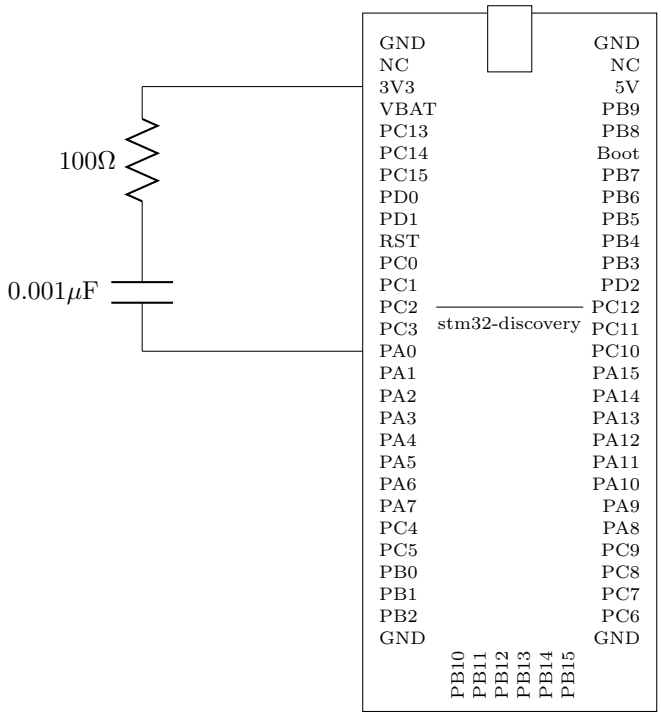



Figure 11.7: De-bounce Circuit

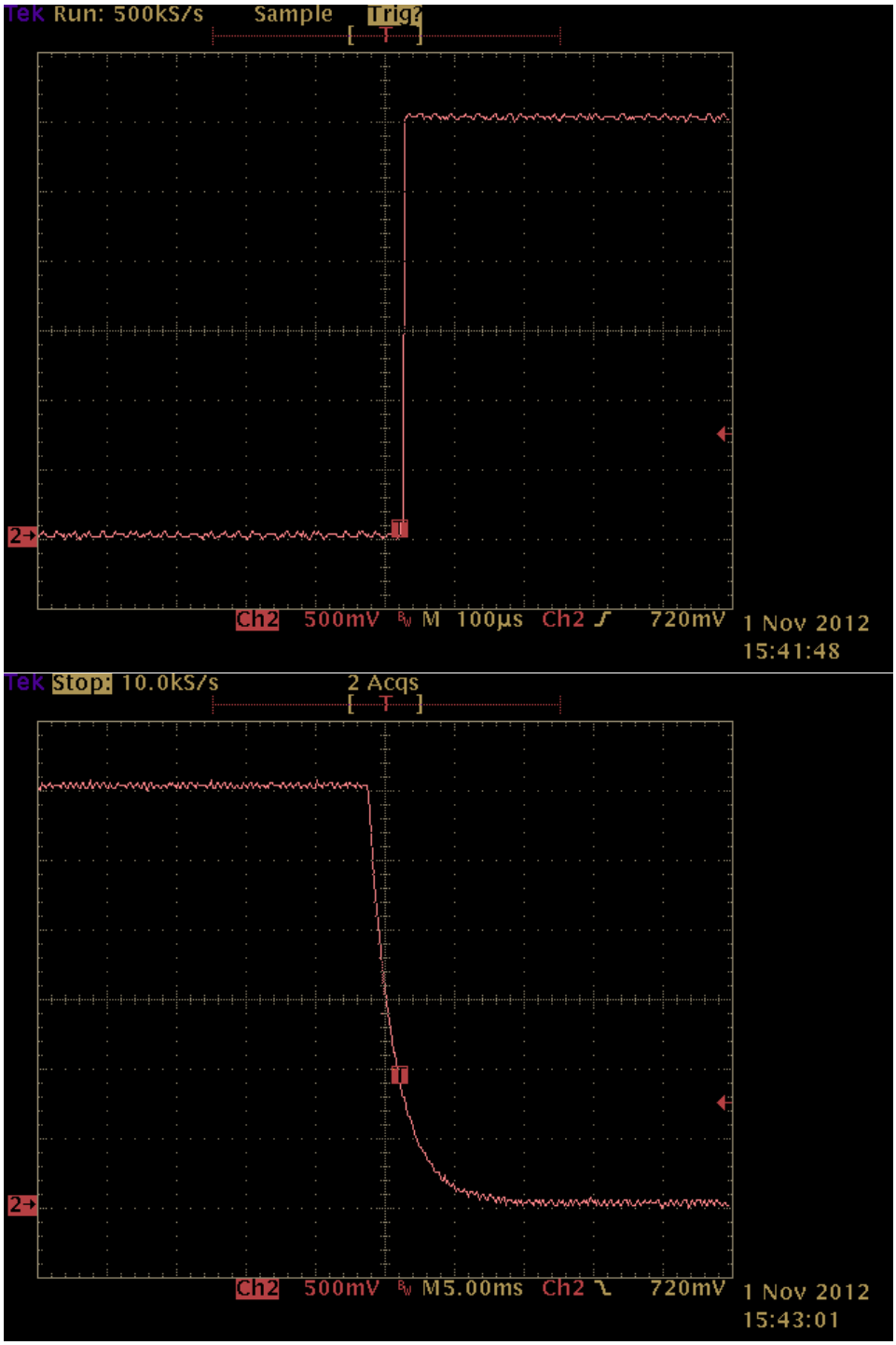





Figure 11.8: Button press/release (filtered)

# Chapter 12

# DMA: Direct Memory Access

In this chapter we discuss the use of direct memory access (DMA) to relieve the processor of the costs of transferring blocks of data between memory and peripherals. Consider the following idiom where a block data is read from a peripheral by repeatedly waiting for a status flag and then reading an item from the peripheral.

```
for (i = 0; i < N; i++) {
  while(flagBusy);
  buf[i] = peripheralRegister;
}
```

We have seen this with serial communication (Section 5.1), SPI communication (Listing 6.3), and will see it again with I2C communication (Figure 9.3). This approach, called software polling, has three limitations. First, the processor is tied up during the transfer and cannot perform other tasks; ideally with a large data transfer (consider reading a data sector from an SD card), the transfer could be kicked off and the processor freed to perform other work while the transfer is realized. Second, the actual transfer rate is lower than the underlying hardware might permit. Finally, it is difficult to achieve tight timing bounds, for example, audio streaming depends upon the data samples to be transferred at a constant rate.

To see rather dramatically the differences in performance, consider the two Logic screen captures showing the SPI transfers when filling and 7735 LCD with a solid color shown in Figure 12. The upper capture measures the time for transferring 2 pixels, while the lower capture measures the time for transferring 128 pixels. The theoretical peak is $12x10^6/16 = 750,000 pixels/second$ for a 12 MHz SPI clock. Without DMA, our trans-

fer rate is $1/20x10^{-6} = 50,000 pixels/second$, while with DMA, our transfer rate is $128/17x10^{-4} = 735,000 pixels/second$. Of course, with DMA, there is unavoidable overhead between blocks. Thus, there is a tradeoff between throughput and memory space (larger blocks yield higher throughput).

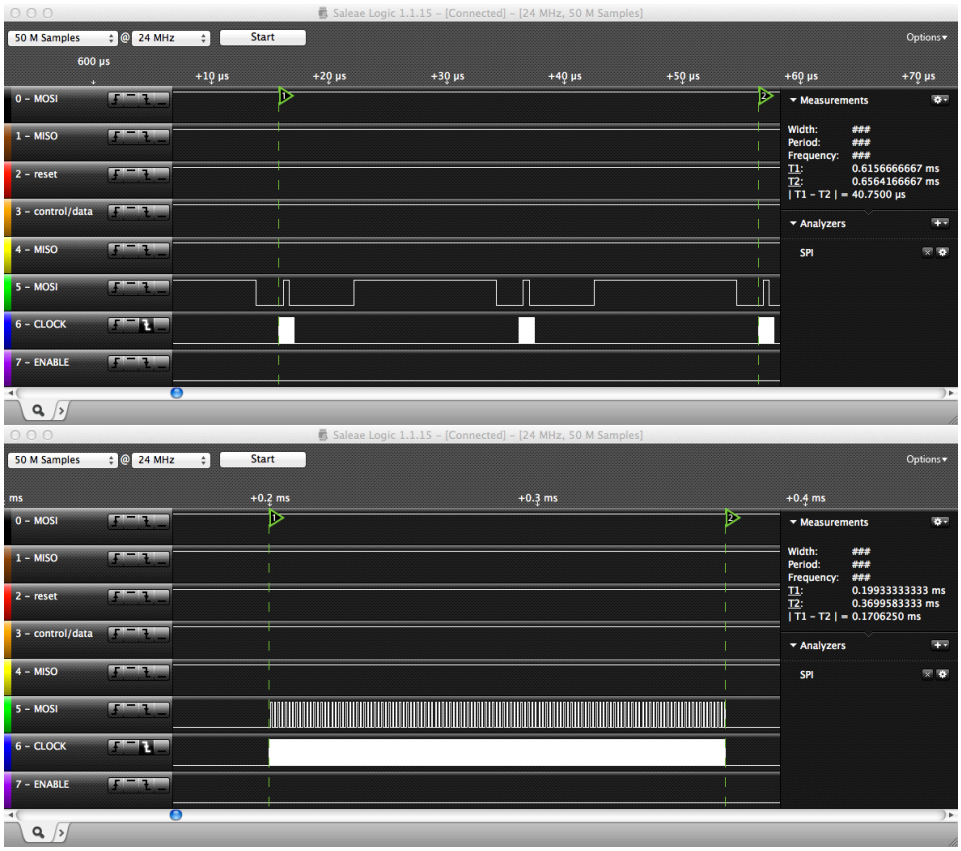



Figure 12.1: Color Fill with (lower) and without (upper) DMA

DMA is implemented in processors with dedicated hardware devices. These devices share the memory bus and peripheral buses with the processor (CPU) as illustrated in Figure 12. In this diagram, the DMA device reads from memory over the memory bus and writes to a peripheral over the peripheral bus. The situation with the STM32 is somewhat more complicated because there are multiple peripheral buses, but the principle is the same.

Only one device can use a bus at any moment, so DMA transfers do impact the CPU. However, consider that the peak rate for the SPI device

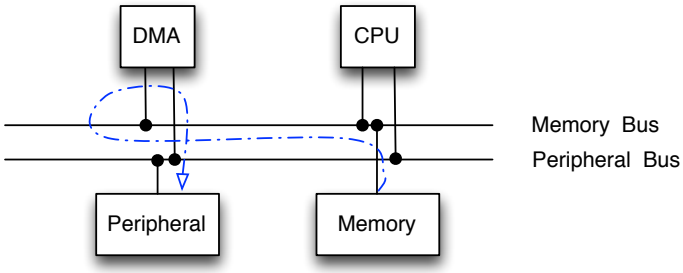



Figure 12.2: DMA Transfer

is 750,000 transfers/second while the memory bus of the STM32 Value line device can support 24,000,000/5 RAM transfers/second (each transfer takes 5 bus cycles). Thus, our block transfer consumes roughly 15% of the memory bus cycles. The STM32 architecture guarantees that the CPU will not be starved. Furthermore, only a fraction of STM32 instructions directly access RAM memory – the rest simply pull instructions from FLASH which uses a different bus.

## 12.1 STM32 DMA Architecture

The STM32 has two DMA peripherals each of which has multiple independently configurable "channels" (7 for DMA1 and 5 for DMA2). A channel is roughly the hardware realization of a transaction. To initialize DMA between a peripheral and memory it is necessary to configure the appropriate channel. For example, DMA1 channel 2 (3) can be used to receive (transmit) data from (to) SPI1.

Prior to utilizing the DMA peripherals, remember to enable their clocks ! For example,

```
RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE);+
```

Configuring a channel consists of setting appropriate parameters through a `DMA_InitTypeDef` structure:

```
typedef struct
{
  uint32_t DMA_PeripheralBaseAddr
  uint32_t DMA_MemoryBaseAddr;
```

```
  uint32_t DMA_DIR;
  uint32_t DMA_BufferSize;
  uint32_t DMA_PeripheralInc;
  uint32_t DMA_MemoryInc;
  uint32_t DMA_PeripheralDataSize;
  uint32_t DMA_MemoryDataSize;
  uint32_t DMA_Mode;
  uint32_t DMA_Priority;
  uint32_t DMA_M2M;
}DMA_InitTypeDef;
```

The parameters include the peripheral base address (e.g. `SPI1->DR`), the memory address buffer, the transfer direction, the buffer size, etc. Once the DMA channel is initialized, it must be enabled. The STM32 firmware provides peripheral specific commands to enable a DMA transaction. The following code fragment illustrates this for transmitting data via SPI utilizing DMA.

```
DMA_Init(txChan, &DMA_InitStructure);
DMA_Cmd(txChan, ENABLE);
SPI_I2S_DMACmd(SPIx, SPI_I2S_DMAReq_Tx, ENABLE);
```

Multiple DMA channels can be initialized simultaneously. Indeed, for SPI the natural configuration utilizes DMA for both transmission and reception simultaneously. Completion of a DMA request is detected through an appropriate flag. We will examine these details more in Section 12.2.

The DMA channels provided by the STM32 are each associated with specific peripherals (See [20] for complete documentation). For example DMA1 channel 1 supports ADC1, TIM2_CH3, and TIM4_CH1. In designing a system with the STM32, it is important to be cognizant of potential resource conflicts. For example DAC_Channel1 (digital analog converter) requires access to DMA1 Channel 3 which is also used for SPI1_TX. Thus it is not possible to simultaneously transmit on SPI1 and output on DAC_Channel1 using DMA. I had initially developed the SPI examples using SPI1; however, later I found that I needed to utilize the DAC for audio output which prompted a switch to SPI2.

## 12.2 SPI DMA Support

A complete SPI 8-bit data receive routine utilizing DMA is illustrated in Listings 12.1 and 12.2. The DMA initialization structure is configured as follows.

```
static int dmaRcvBytes(void *rbuf, unsigned count)
{
  DMA_InitTypeDef DMA_InitStructure;
  uint16_t dummy[] = {0xffff};

  DMA_DeInit(DMA1_Channel2);
  DMA_DeInit(DMA1_CHannel3);

  // Common to both channels

  DMA_InitStructure.DMA_PeripheralBaseAddr =
      ↪(uint32_t)(&(SPI1->DR));
  DMA_InitStructure.DMA_PeripheralDataSize =
      ↪DMA_PeripheralDataSize_Byte;
  DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_Byte;
  DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;
  DMA_InitStructure.DMA_BufferSize = count;
  DMA_InitStructure.DMA_Mode = DMA_Mode_Normal;
  DMA_InitStructure.DMA_Priority = DMA_Priority_VeryHigh;
  DMA_InitStructure.DMA_M2M = DMA_M2M_Disable;

  // Rx Channel

  DMA_InitStructure.DMA_MemoryBaseAddr = (uint32_t)rbuf;
  DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;
  DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralSRC;

  DMA_Init(DMA1_Channel2, &DMA_InitStructure);

  // Tx channel

  DMA_InitStructure.DMA_MemoryBaseAddr = (uint32_t) dummy;
  DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Disable;
  DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralDST;

  DMA_Init(DMA1_channel3, &DMA_InitStructure);
  /* ... */
```

Listing 12.1: SPI DMA Receive (Part 1)

- Peripheral base address set to SPI data register.
- Peripheral and memory data size set to 8 bits (byte).
- Peripheral increment disabled – all data read/written from a single DR register.

```
  // Enable channels

  DMA_Cmd(rxChan, ENABLE);
  DMA_Cmd(txChan, ENABLE);

  // Enable SPI TX/RX request

  SPI_I2S_DMACmd(SPIx, SPI_I2S_DMAReq_Rx | SPI_I2S_DMAReq_Tx,
      ↪ENABLE);

  // Wait for completion

  while (DMA_GetFlagStatus(DMA1_FLAG_TC2) == RESET);

  // Disable channels

  DMA_Cmd(rxChan, DISABLE);
  DMA_Cmd(txChan, DISABLE);

  SPI_I2S_DMACmd(SPIx, SPI_I2S_DMAReq_Rx | SPI_I2S_DMAReq_Tx,
      ↪DISABLE);

  return count;
}
```

Listing 12.2: SPI DMA Receive (Part 2)

- Buffer size set to number of data to be transferred (count).
- Normal DMA mode
- High priority

The code then configures the memory side and initializes the receive channel followed by reconfiguring the memory side and configuring the transmit channel. For uni-directional read operations, transmit data are provided via a "dummy" buffer which is read repeatedly during the transfer (DMA_MemoryInc_Disable).

Once both channels are configured, they are enabled, a SPI DMA request is made, and finally the code waits for a completion flag. Notice that in this implementation, the processor sets up the transfer and then waits for completion, but this still involves busy waiting. An alternative approach (not

considered here) is to enable a DMA interrupt and define an appropriate handler (e.g. DM1_Channel1_IRQHandler). The main code could then return and await an appropriate completion event. I postpone discussion of such an architecture until I have the opportunity to introduce an RTOS.

### Exercise 12.1 SPI DMA module

Implement a generic SPI exchange routine:

```
static int xchng_datablock(SPI_TypeDef *SPIx, int half, const void
    ↪*tbuf, void *rbuf, unsigned count)
```

Your configuration should utilize the DMA channels for either SPI1 or SPI2 as defined at call time. The transfer size should be configured as 8-bits or 16-bits as defined by the parameter `half`. It should be possible to utilize this routine as read/write/exchange as needed.

You should then create a new SPI module (call it spidma.c) which implements the interface in Figure 6.1 and utilizes DMA for all transfers longer than 4 data items (shorter transfers should be performed without DMA). This module should be derived your polling-based spi.c module.

### Exercise 12.2 Display BMP Images from Fat File System

Write an application that reads BMP images (see `http://en.wikipedia.org/wiki/BMP_file_format`) from an SD card and displays them on the 7735 screen. Note that BMP images are usually stored from bottom to top so you will need to appropriately set the direction when using the 7735 interface. To help you get started, a simple Linux program that parses BMP files is illustrated in Listings 12.3 and 12.4. You can manipulate images in various formats (e.g. resize, rotate) with the Linux program `convert`. Your program should probably check the size of BMP images it attempts to display and reject those which are not 128x160 and 24bit color. You program will have to convert from 24 bit color to 16-bit color. The conversion is relatively easy – for each RGB byte, select the lower 5 or 6 bits (RGB = 565) and pack into a single 16-bit color byte. Finally, your program should cycle through the set of BMP files in the root directory of an SD card. Note – remember that FAT16 file systems use 8.3 naming – that is 8 characters followed by a 3 character previx !

To obtain the benefits of DMA you should read a block of pixels into a buffer, convert them and pack in second buffer, and then write them out. Experiment with different block sizes 16,32,64,128 pixels and measure how

long it takes to load and display a file. You can easily utilize the system timer interrupt to measure delays. For example, add a variable that is incremented on every interrupt; clear this variable before reading and writing an image. Compare your results with and without DMA.

Linux provides several tools for manipulating images. To resize a JPEG image to the 128x160 geometry of the lcd:

```
convert -resize 128x160! input.jpg output.jpg
```

Conversion to a BMP file is accomplished with a pair of tools

```
jpegtopnm output.jpg |ppmtobmp > output.bmp
```

```
#include <stdint.h>

struct bmpfile_magic {
  unsigned char magic[2];
};

struct bmpfile_header {
  uint32_t filesz;
  uint16_t creator1;
  uint16_t creator2;
  uint32_t bmp_offset;
};

typedef struct {
  uint32_t header_sz;
  int32_t  width;
  int32_t  height;
  uint16_t nplanes;
  uint16_t bitspp;
  uint32_t compress_type;
  uint32_t bmp_bytesz;
  int32_t  hres;
  int32_t  vres;
  uint32_t ncolors;
  uint32_t nimpcolors;
} BITMAPINFOHEADER;

struct bmppixel { // little endian byte order
  uint8_t b;
  uint8_t g;
  uint8_t r;
};
```

Listing 12.3: BMP File Structures

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

struct bmpfile_magic magic;
struct bmpfile_header header;
BITMAPINFOHEADER info;

main(int argc, char *argv[])
{
  int f;
  if (argc > 1){
      if ((f = open(argv[1], O_RDONLY)) == -1)
      {
          perror(``problem opening file'');
          return 1;
      }
      read(f, (void *) &magic, 2);
      printf(``Magic %c%c\n'', magic.magic[0], magic.magic[1]);
      read(f, (void *) &header, sizeof(header));
      printf(``file size %d offset %d\n'', header.filesz,
          ↪header.bmp_offset);
      read(f, (void *) &info, sizeof(info));
      printf(``Width %d Height %d, bitspp %d\n'', info.width,
           info.height, info.bitspp);
      close(f);
  }
}
```

Listing 12.4: Parsing BMP Files

# Chapter 13

# DAC : Digital Analog Converter

The STM32 F1xx parts have a 12-bit digital-analog-converter (DAC) module with two independent output channels – DAC1 (pin PA4) and DAC2 (pin PA5). The channels can be configured in 8-bit mode or 12-bit mode and the conversions can be done independently or simultaneously. Simultaneous mode can be used where two independent but synchronized signals must be generated – for example, the left and right channels of stereo audio. It is often important that the analog output is update at precise instants, sometimes controlled by external hardware – thus the conversion process can be triggered by timers or external signals. Finally, a common use for DAC hardware is to generate a time varying signal. Where the sample rate is high, it is impractical to control the conversion process entirely through application software or even interrupt handlers. Thus, each DAC channel has DMA capability which can be controlled by the trigger signal.

In this chapter we will explore the use of a single DAC channel. We will start with the simplest case – direct software control of a analog output – which is useful for creating slow moving signals that do not have to be synchronized in time. We will then explore the generation of time-synchronized signals; first with DAC module provided triangle wave generator, and then interrupt driven code to produce a sine wave. Interrupt driven operation does not scale well to high update rates – eventually, all of the processor cycles are required just to service the interrupt. Thus, we will examine the use of DMA to "feed" the DAC, with a lower frequency interrupt used to update the DMA buffer. Finally, we will define an exercise to read audio files from an SD card and play them, via DMA, to an external audio amplifier driven by the DAC.

**Warning:** The preliminary exercises in this chapter cannot easily be completed without access to an oscilloscope to display waveforms. The final exercise, an audio player, can be completed without an oscilloscope but may be harder to debug without one.

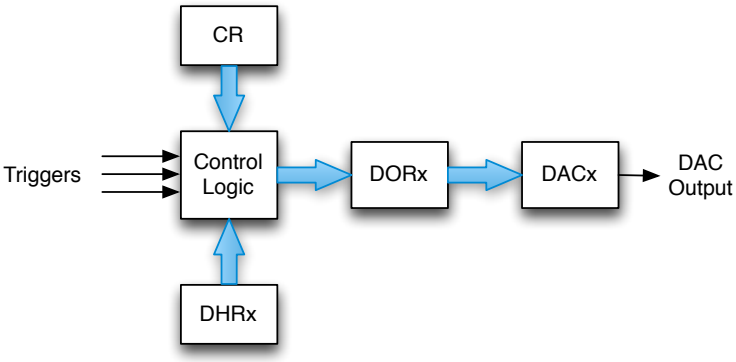



Figure 13.1: DAC Channel x

A simplified block diagram of one DAC channel is illustrated in Figure 13. Each channel has separate control logic which is configured through a single control register (CR). Data to be converted by channel x are written to data holding register (DHRx). In response to a trigger event, DHRx is transferred to the data output register (DORx) and, after a settling time, the corresponding analog value appears at the output. The DAC output voltage is linear between 0 and $V_{REF+}$ (3.3V on the discovery board) and is defined by the following equation:

$$DACoutput = V_{REF+} \times \frac{DOR}{4095}$$

Trigger events can include the trigger signals from various timers (TIM2-TIM7, TIM15), an external interrupt (EXTI line 9), and software. It is also possible to configure the DAC without a trigger in which case DHRx is automatically copied to the DORx after a single clock cycle delay.

### Exercise 13.1 Waveform Generator

Write a C program to generate a sine wave with minimum and maximum values of 512 and 1536 where a full cycle consists of 100 samples. The output of your program should have the form:

```
int16_t a441[] = {
    ...
};
```

In order to simplify data alignment and synchronous output, the software interface for the two data holding registers is through a set of registers each of which addresses a specific alignment issue. The various cases are illustrated in Figure 13. For each of the two DAC channels there are registers for left and right-aligned 12-bit data as well as 8-bit data. Writing to these registers affects the appropriate DHRx register as illustrated. To enable synchronous output, there are three interface registers (DHR12RD, DHR12LD, and DHR8RD) for each of the three data alignment cases. Notice that for the case of 8-bit data, the data fill bits 11..4 of the DHR (the high order bits) which preserves the full 0..VREF range of the analog output.

In addition to converting user supplied data, the DAC channels each independently support noise-wave and triangle-wave generation. In these modes, data to be converted are generated in response to the trigger event. The output waveforms can be useful both for testing the DAC and for testing external hardware.

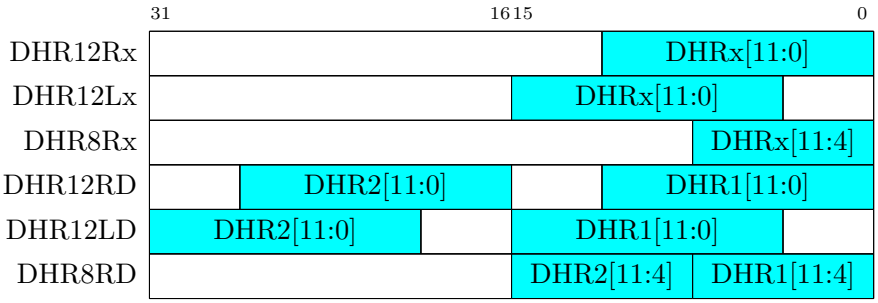



Figure 13.2: Data Holding Registers

### Exercise 13.2 Application Software Driven Conversion

A common use for DAC hardware is to generate an analog output which changes infrequently and which does not need to be synchronized to any signal. Configuration follows the pattern for all previous devices – enable clocks, configure pins, configure device, enable device. In the basic use, the default configuration for the DAC is adequate, although we choose to enable an output

buffer which provides a bit more electrical drive at the expense of speed as illustrated in Listing 13.1.

Somewhat confusingly, the pin is configured as "analog in". The name is a bit misleading because the DAC module drives the pin when configured. However, this configuration does perform the important function of disconnecting any digital input buffer ! Once a DAC is initialized, data may be written using the following commands (defined in `stm32f10x_dac.[ch]` where align is one of `DAC_Align_12b_R`, `DAC_Align_12b_L`, `DAC_Align_8b_R`.

```
DAC_SetChannel1Data(align,value);
DAC_SetChannel2Data(align,value);
```

Now write a program that, on a 1ms interval, output the 12-bit values corresponding to the sin wave you created above (a full cycle consists of 100 samples). Test the output by examining on an oscilloscope.

```
  GPIO_InitTypeDef            GPIO_InitStructure;
  DAC_InitTypeDef             DAC_InitStructure;

  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_DAC, ENABLE);

  GPIO_StructInit(&GPIO_InitStructure);
  GPIO_InitStructure.GPIO_Pin =  GPIO_Pin_4;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN;
  GPIO_Init(GPIOA, &GPIO_InitStructure);

  // DAC channel1 Configuration

  DAC_StructInit(&DAC_InitStructure);
  DAC_InitStructure.DAC_OutputBuffer = DAC_OutputBuffer_Enable;
  DAC_Init(DAC_Channel_1, &DAC_InitStructure);

  // Enable DAC

  DAC_Cmd(DAC_Channel_1, ENABLE);
```

Listing 13.1: DAC Initialization

## Exercise 13.3 Interrupt Driven Conversion

Forcing the main application program to maintain precise timing is not a robust solution. First, it ties the application program to a tight inner loop,

and second, in the presence of interrupts, we cannot guarantee that the data are output precisely a the desired delay. The timing "jitter" introduced can be a serious problem in some applications. For example, with audio, jitter can be heard in the form of distortion.

An alternative approach is to configure a timer to generate interrupts at the desired conversion frequency and using an interrupt handler to write the DHR register. Once a timer is introduced into the solution, we can also use the timer interrupt to trigger the transfer from the DHR to the DOR. This insulates the DAC timing from any possible delays due to other interrupts. As long as the interrupt handler is allowed to execute before the next timer "tick" there will be no jitter in the output signal.

In this exercise we will utilize TIM3 both to trigger the DAC transfer and to call an interrupt handler that will be responsible for reloading the DHR. You may modify the previous exercise as follows.

- Configure TIM3 to update at 44.1 kHz – see Chapter 10 to review how this is done.
- Configure the TIM3 output trigger so that it occurs on update events:
  ```
  TIM_SelectOutputTrigger(TIM3, TIM_TRGOSource_Update);
  ```
- Configure the NVIC to enable interrupt channel `TIM3_IRQn` with the highest priority (0 for both sub-priority and preemption priority) – see Chapter 11 to see how this is done.
- Change the trigger in the DAC initialization code:
  ```
  DAC_InitStructure.DAC_Trigger = DAC_Trigger_T3_TRGO;
  ```
- Write a suitable interrupt handler.
  ```
  void TIM3_IRQHandler(void){
  }
  ```

Test your program first using an oscilloscope and then connect it to the external speaker module. You should hear an "A" (actually, 441Hz) if everything is working properly.

Note that when configuring the timer, you do not need to set the prescaler to ensure that the timer period is a particular number of cycles as we did for the PWM. In this example, we are only concerned with the frequency of the update events – not how many clock ticks occurred between update events.

## 13.1 Example DMA Driven DAC

While an interrupt driven DAC has fewer jitter issues, ultimately, the interrupt handler will consume all the available cycles. With the use of DMA we can reduce the software overhead by a factor proportional to the buffer size that we are willing to allocate. The DMA hardware handles the details of writing individual samples to the DAC at the timer rate while an interrupt handler is responsible for refreshing the buffer. As we shall see, with a N-item buffer, we can reduce our interrupt rate by a factor of N/2.

This example builds upon Exercise 13.3. The major changes are:

- Initialize a DMA channel (DMA1 Channel 3)
- Initialize the NVIC for an interrupt handler
- Create an interrupt handler

We continue to use TIM3 to drive the conversion process; however, rather than calling an interrupt handler to generate the next datum, the DMA device provides this datum. We assume that you have generated a 100 item array with the A440 tone. For this example, we will create a buffer of the same size – initially a perfect clone.

```
for (i = 0; i < A440LEN; i++)
    outbuf[i] = a440[i];
```

The DAC interrupt handler is responsible for refilling this buffer in alternating halves (Listing 13.2). In this example, the refilling action is not particularly interesting since we are simply re-copying the A440 tone into our output buffer. However, the key idea is that the interrupt handler is called with the DMA transfer is completed (`DMA1_IT_TC3`) and "half completed" (`DMA_IT_HC3`) – each case fills a different half of the buffer. This approach provides the greatest resilience to potential timing issues.

Configuration is somewhat more complicated because of the need to initialize a DMA channel (Listing 13.3). Notice that the peripheral register is for DHR12R1, and that the DMA is operating in "circular mode" which means that it continuously pulls from a single memory buffer. The DMA channel is configured to generate interrupts at the half and full completion marks. Another notable configuration step is the configuration of the DAC to utilize DMA `DAC_DMACmd(DAC_Channel_1, ENABLE)`.

```
int completed; // count cycles

void DMA1_Channel3_IRQHandler(void)
{
  int i;
  if (DMA_GetITStatus(DMA1_IT_TC3)){      // Transfer complete
    for (i = A440LEN/2; i < A440LEN; i++)
      outbuf[i] = a440[i];
    completed++;
    DMA_ClearITPendingBit(DMA1_IT_TC3);
  }
  else if (DMA_GetITStatus(DMA1_IT_HT3)){ // Half Transfer complete
    for (i = 0; i < A440LEN/2; i++)
      outbuf[i] = a440[i];
      DMA_ClearITPendingBit(DMA1_IT_HT3);
    }
}
```

Listing 13.2: DMA Interrupt Handler

### Exercise 13.4 Audio Player

Create an audio player with the interface illustrated in Listing 13.4. The player should be based on your previous code and the example in Section 13.1, but with the following differences: the data size should be 8-bits, the data rate should be configurable and the interrupt handler should simply set two the two flags `audioplayerHalf` and `audioplayWhole` when the DMA half and whole transfers, respectively, are complete. The job of keeping the audio buffer `Audiobuf` full is pushed to the application which must poll these two flags, refill the bottom or top half of the buffer as appropriate, and reset the flag; later, when we examine a real-time operating system, this interface will adapt easily to a thread based system. The "init" function should configure the timer, DMA channel, DAC, and interrupt; but it should not enable the hardware. The "start" function should enable the hardware – before calling start, the application must first fill the data buffer. The "stop" function should disable the hardware. Once a sound is completely played the application needs to stop the player. Test this interface with a sine wave. Next, we'll want to read audio files from an SD Card.

An audio player is much more interesting if it can read "standard" audio files. In the following we will describe a subset of the WAV file format which can be generated by most audio software. We only consider the case for PCM encoding with a single audio channel (mono). If you have stereo audio files

```
// DMA Channel 3 Configuration

RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE);
DMA_DeInit(DMA1_Channel3);

DMA_StructInit(&DMA_InitStructure);
DMA_InitStructure.DMA_PeripheralBaseAddr = &DAC->DHR12R1;
DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralDST;
DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;
DMA_InitStructure.DMA_PeripheralDataSize =
    ↪DMA_PeripheralDataSize_HalfWord;
DMA_InitStructure.DMA_BufferSize = A440LEN;
DMA_InitStructure.DMA_MemoryBaseAddr = (uint32_t) outbuf;
DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;
DMA_InitStructure.DMA_MemoryDataSize =
    ↪DMA_MemoryDataSize_HalfWord;
DMA_InitStructure.DMA_Mode = DMA_Mode_Circular;
DMA_InitStructure.DMA_Priority = DMA_Priority_High;
DMA_InitStructure.DMA_M2M = DMA_M2M_Disable;

DMA_Init(DMA1_Channel3, &DMA_InitStructure);

//  Enable Interrupts for complete and half transfers

DMA_ITConfig(DMA1_Channel3, DMA_IT_TC | DMA_IT_HT , ENABLE);

NVIC_InitStructure.NVIC_IRQChannel = DMA1_Channel3_IRQn;
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
    ↪//HIGHEST_PRIORITY;
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
NVIC_Init(&NVIC_InitStructure);

// Enable everything

DMA_Cmd(DMA1_Channel3, ENABLE);
DAC_Cmd(DAC_Channel_1, ENABLE);
DAC_DMACmd(DAC_Channel_1, ENABLE);
TIM_Cmd(TIM3, ENABLE);
```

Listing 13.3: DMA Configuration

or files in other formats or with different encodings, the Linux program "sox" provides a powerful conversion utility. You should write a parser for this subset of WAV to test on a desktop machine – it is sufficient to print out the key fields when parsing a file.

```
#ifndef PLAYER_H
#define PLAYER_H

#define AUDIOBUFSIZE 128

extern uint8_t Audiobuf[];
extern int audioplayerHalf;
extern int audioplayerWhole;

audioplayerInit(unsigned int sampleRate);
audioplayerStart();
audioplayerStop();

#endif
```

Listing 13.4: Audio Player Interface

There are many references to WAV files available on the WWW. A good one can be found at `http://www-mmsp.ece.mcgill.ca/documents/audioformats/wave/wave.html`. A WAV file consists of a (recursive) sequence of chunks. Every chunk except the first has the following form:

| Field | Length (bytes) | Content |
|---|---|---|
| ckID | 4 | Chunk ID (e.g. "RIFF") |
| cksize | 4 | Chunk size n |
| body | n | Chunk contents |

With the exception of the ckID field, which is a sequence of four characters in big-endian order, all other fields are in little-endian order. The WAV files we consider have the following sequence: Master (RIFF) chunk, format chunk, 0 or more other chunks, data chunk.

To parse WAV files we need to be able to read chunk headers, parse the three chuck types of interest, and skip uninteresting chunks. The master RIFF chunk has the following form:

| Field | Length (bytes) | Content |
|---|---|---|
| ckID | 4 | Chunk ID: "RIFF" |
| cksize | 4 | Chunk size 4+n |
| WAVEID | 4 | WAVE ID: "WAVE" |
| WAVE chunks | n | Chunk contents |

The Format chunk has the following form. There are three variations for this, but we only need to parse the common portion; however, your code

should skip any ignored bytes !

| Field | Length (bytes) | Content |
|---|---|---|
| ckID | 4 | Chunk ID: “fmt ” |
| cksize | 4 | Chunk size 16 or 18 or 40 |
| wFormatTag | 2 | 0x0001 for PCM data |
| nChannels | 2 | Number of channels – 1 for mono |
| nSamplesPerSec | 4 | Sampling rate (per second) |
| nAvgBytesPerSec | 4 | Data rate |
| nBlockAlign | 2 | Data block size (bytes) |
| wBitsPerSample | 2 | Bits per sample – 8 |
| | 0-24 | ignore |

The key pieces of information are the format (must be 1 for PCM), number of channels (must be 1 for mono), sampling rate (to be configured in the player), bits per sample (must be 8).

The Format chunk may be followed by 0 or more non-data chunks which your parser should skip. The final chunk we consider is the data chunk:

| Field | Length (bytes) | Content |
|---|---|---|
| ckID | 4 | Chunk ID: “data” |
| cksize | 4 | n |
| sampled data | n | Samples |
| pad byte | 0 or 1 | Pad if n is odd |

To write your parser here are some helpful data structures:

```
#define RIFF 'FFIR'
#define WAVE 'EVAW'
#define fmt  ' tmf'
#define data 'atad'

struct ckhd {
  uint32_t ckID;
  uint32_t cksize;
};

struct fmtck {
  uint16_t wFormatTag;
  uint16_t nChannels;
  uint32_t nSamplesPerSec;
  uint32_t nAvgBytesPerSec;
  uint16_t nBlockAlign;
  uint16_t wBitsPerSample;
};
```

It is convenient to compare the chunk IDs against strings; however, the conventional C string has a null termination byte. An alternative, shown in this preceding code fragment is a multi-byte character (defined in reverse order to handle endianess !).

Your parser should take a WAV file name as argument, open and parse the file. As mentioned, when parsing the file, your parser should print out the relevant field. If you use the "open", "close", and "read" system commands, then skipping over uninteresting chunks is easily accomplished:

```
lseek(fid, skip_amount , SEEK_CUR);
```

Once your parser is working, it's time to create an audio player that can read a WAV file off an SD card and play it. The fat file system described in Chapter 8 has read (`f_read`), and lseek (`f_lseek`) commands with slightly different interfaces than the Linux equivalent. You should modify your audio player to open a particular WAV file (you'll need to copy a suitable file to an SD card) from an SD card, parse the header, initialize the player, and then "play" the file – here's an example of how I handled the `audioplayerHalf` event (next is the number of bytes to be copied which is generally half the buffer size except at the file end).

```
if (audioplayerHalf) {
  if (next < AUDIOBUFSIZE/2)
     bzero(Audiobuf, AUDIOBUFSIZE/2);
  f_read(&fid, Audiobuf, next, &ret);
     hd.cksize -= ret;
  audioplayerHalf = 0;
}
```

# Chapter 14

# ADC : Analog Digital Converter

The dual of a DAC is an ADC (analog digital converter) which converts analog signals to digital values. The STM32 processors include one or more ADC peripherals – there is one on the medium density value line parts. The STM32 ADC uses successive approximation – the ADC has the ability to generate a discrete set of voltages and to compare these against a sampled input voltage; it essentially performs binary search to find the best approximation. For 12 bits of accuracy, the STM32 takes at least 14 cycles of the ADC clock (a multiple of the system clock) – the extra 2 cycles are overhead due to sampling. Thus, With a 12 MHz ADC clock, the STM32 ADC can perform a sample in slightly more than $1\mu sec$.

Although the STM32 VL component has a single ADC, it can support multiple analog inputs. The basic architecture is illustrated in Figure 14. On the discovery board, PA0-PA7, PB0-PB1, and PC0-PC5 may all be used as analog inputs which can be multiplexed to the ADC. The ADC may be configured to sample any subset of these inputs in succession. There are two basic modes of operation – single and continuous conversion. With single conversion, once the ADC is triggered, it converts a single input and stores the result in its data register (DR). The trigger may either come from software or signal such as a timer. In continuous mode, the ADC starts another conversion as soon as it finishes one. The ADC may also operate in scan mode where a set of inputs to be scanned is configured. A single conversion is performed for each configured input in succession. Scans may also be continuous in the sense that a new scan begins as soon as one is completed.

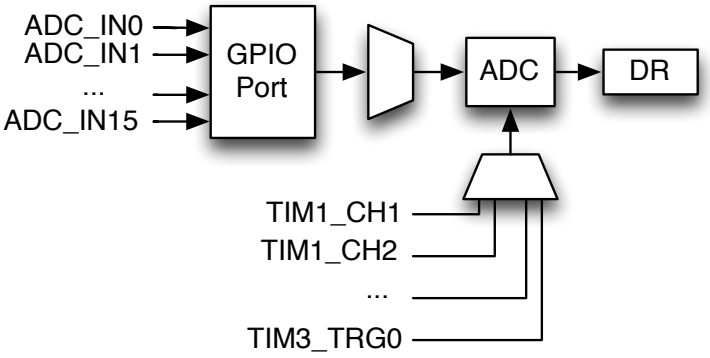



Figure 14.1: Analog Digital Converter

The ADC has a single data register, so when scanning multiple analog inputs, it is essential that the data be read between samples. This can be accomplished through polling – software can monitor a flag, interrupts, or DMA. Interrupts and DMA may be triggered at the end of each conversion. In this chapter we show how to use the ADC to continuously sample a single input, to use a timer to trigger conversion of a single channel and an ADC interrupt to read the result, and how to use DMA to handle conversions in the time triggered case. We also present an exercise to create a voice recorder that saves the resulting WAV files on an SD card.

It is important to note that our presentation ignores several additional features of the STM32 ADC (for example "injected channels") that are best understood by reading the reference manual.

## 14.1 About Successive Approximation ADCs

To fully understand the STM32 ADC it is helpful to have some background in how successive approximation ADCs operate. Consider Figure 14.1 which illustrates a typical ADC. At its heart is a digital analog converter (DAC). To begin a conversion, a hardware control "captures" a sample of input voltage ($V_{ain}$) – shown here with some external resistance $R_{ain}$ to be discussed in the sequel. Once the input ($V_{samp}$) is captured, the controller generates a sequence of digital approximations $D(v_{est})$ and checks each by converting the approximation to an analog signal which is then compared against the sampled input. The sequence of approximations corresponds to

a binary search – MSB to LSB. Once the best approximation is found, the digital value is loaded into a data register ($DR$). For a N-bit approximation, the approximation process requires N iterations.

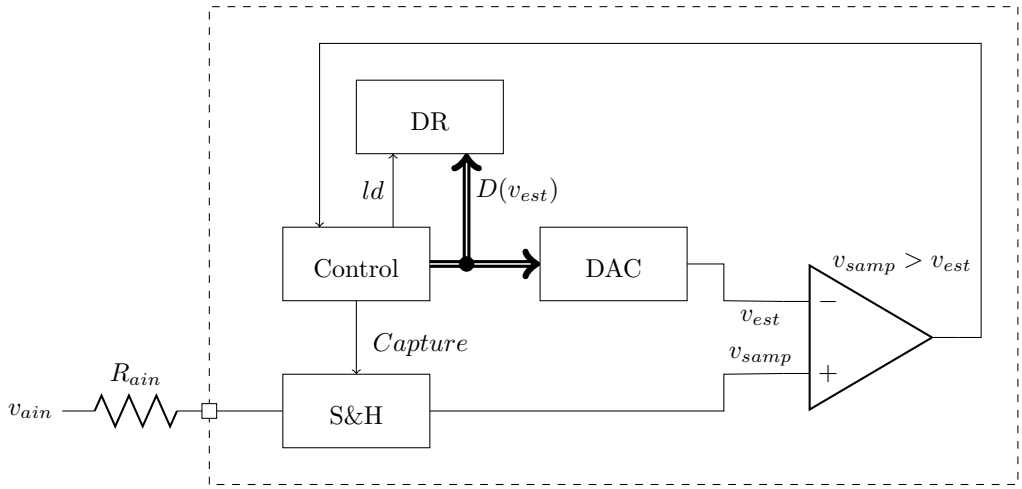



Figure 14.2: Successive Approximation ADC

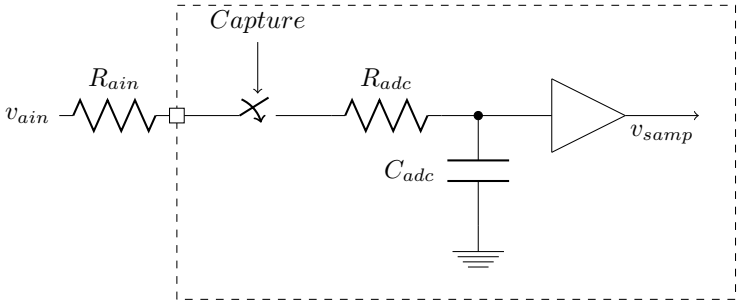



Figure 14.3: Sample and Hold

The sample and hold circuit operates by charging an internal capacitor as illustrated in Figure 14.1. During the capture phase, a switch (transistor) is closed and the internal capacitor $C_{adc}$ is allowed to charge/discharge to voltage $v_{ain}$. This is where the external resistance comes into play. If the external voltage source has "high-impedance", it cannot produce a large current and hence charging the capacitor may take a long time. In contrast, a

low-impedance source can charge the capacitor quickly. Thus, the time required to capture an input voltage is dependent on the external circuit; in the STM32, the capture time is configurable from 1.5 cycles to 239.5 cycles to accommodate differences in inputs. For high speed conversion, it may be necessary to use an external op-amp to buffer an input in order to convert a high-impedance source into a low-impedance input. Where speed is not an issue, it is best to choose the longest sample time that is feasible.

To understand the impact of sample time on accuracy, consider the graph in Figure 14.1. The time required to charge the internal capacitor $C_{adc}$ is dependent upon the internal and external impedances –

$$t_c = C_{adc} * (R_{adc} + R_{ain})$$

with the capacitor voltage at time $t$ equal to

$$v_{samp} = v_{ain} \times (1 - e^{-t/t_c})$$

In general, it is necessary to wait a multiple of $t_c$ to achieve reasonable accuracy. For example given $t_c$ we can compute the sample time required to achieve 12 bits of accuracy:

$$\begin{aligned}
\frac{v_{samp}}{v_{ain}} &> 1 - \frac{1}{2^{12}} \\
1 - e^{-t/t_c} &> 1 - 2^{-12} \\
e^{-t/t_c} &\leq 2^{-12} \\
\frac{-t}{t_c} &\leq ln(2^{-12}) \\
\frac{-t}{t_c} &\leq -12 \times ln(2) \\
t &> 8.32 \times t_c
\end{aligned}$$

The model presented is fairly simplistic, but illustrates some of the key ideas. Other factors to be considered are board layout (stray capacitance) reference voltages, signal noise, etc. All this is described in detail in [12]. It is sufficient to conclude by saying that achieving fast and precise analog to digital conversion is complex; however, many applications to not require either high speed or high precision. In such applications, only modest attention to detail is required.

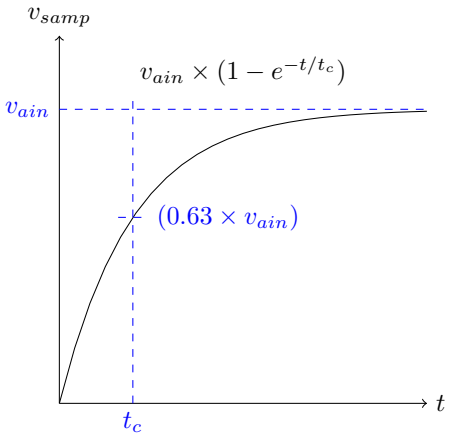



Figure 14.4: Impact of Sample Time on Accuracy

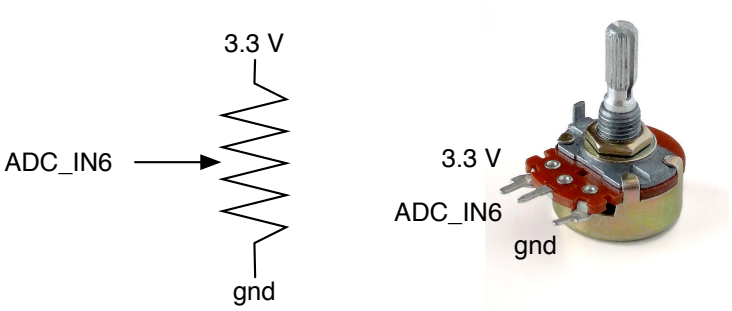



Figure 14.5: Simple Analog Input

### Exercise 14.1 Continuous Sampling

In this example a simple analog source – a potentiometer will be connected to PA6 (ADC_IN6). As illustrated in Figure 14.1 two terminals of the potentiometer should be connected to 3.3 V and GND while the center tap is connected to ADC_IN6. On the left side of the figure is the electrical symbol for a potentiometer and on the right a photograph of a typical potentiometer. A potentiometer is a variable resistor; in this application we use it a voltage divider which can produce any input between 0 and 3.3 V.

The application will continuously read the analog input and, if it is

above 3.3/2 Volts, light the green LED, or turn the LED off otherwise. By now you should know how to do the following:

- Configure the clocks for GPIOA, GPIOC, and the ADC.
- Configure port PA6 as analog input.
- Configure port PC9 as push/pull output.

Configuring the ADC follows a familiar pattern:

```
  ADC_InitStructure.ADC_Mode = ADC_Mode_Independent;
  ADC_InitStructure.ADC_ScanConvMode = DISABLE;
  ADC_InitStructure.ADC_ContinuousConvMode = ENABLE;
  ADC_InitStructure.ADC_ExternalTrigConv =
      ↪ADC_ExternalTrigConv_None;
  ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
  ADC_InitStructure.ADC_NbrOfChannel = 1;

  ADC_Init(ADC1, &ADC_InitStructure);

// Configure ADC_IN6

 ADC_RegularChannelConfig(ADC1, ADC_Channel_6, 1,
     ↪ADC_SampleTime_55Cycles5);

// Enable ADC

  ADC_Cmd(ADC1, ENABLE);
```

Notice that when configuring the channel, we have to select a "sampling time." The minimum for 12 bits of accuracy is a 13.5 cycle sampling time. Choosing the "right" sampling time can be complicated if it is desirable to combine speed and accuracy. Our requirements for this example are pretty loose.

Unlike other peripherals, the ADC needs to be calibrated. The hardware supports automated calibration as the following code illustrates. Again for this example, we can probably dispense with calibration.

```
// Check the end of ADC1 reset calibration register
  while(ADC_GetResetCalibrationStatus(ADC1));
// Start ADC1 calibration
  ADC_StartCalibration(ADC1);
// Check the end of ADC1 calibration
  while(ADC_GetCalibrationStatus(ADC1));
```

The only remaining issue is reading the conversion result:

```
ain = ADC_GetConversionValue(ADC1);
```

It remains to create a working application.

### Exercise 14.2 Timer Driven Conversion

In this exercise we will convert the preceding exercise to be timer driven with an interrupt handler to execute whenever the conversion is complete. We will need to slightly alter the ADC initialization code, configure a timer, and configure the NVIC.

You should configure the NVIC to enable `ADC1_IRQn` – the specific priority is relatively unimportant. Configure TIM3 to generate a TRG0 event every 1ms. Configure the ADC as follows:

```
  ADC_InitStructure.ADC_Mode = ADC_Mode_Independent;
  ADC_InitStructure.ADC_ScanConvMode = DISABLE;
  ADC_InitStructure.ADC_ContinuousConvMode = DISABLE;
  ADC_InitStructure.ADC_ExternalTrigConv =
                            ADC_ExternalTrigConv_T3_TRGO;
  ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
  ADC_InitStructure.ADC_NbrOfChannel = 1;

  ADC_Init(ADC1, &ADC_InitStructure);

  ADC_RegularChannelConfig(ADC1, ADC_Channel_6, 1,
                           ADC_SampleTime_55Cycles5);

  ADC_ITConfig(ADC1, ADC_IT_EOC, ENABLE);

  ADC_ExternalTrigConvCmd(ADC1, ENABLE);

  ADC_Cmd(ADC1, ENABLE);
```

Move your code that toggles the LED to an interrupt handler with the following structure:

```
void ADC1_IRQHandler(void)
{
  // read ADC DR and set LED accordingly

  ADC_ClearITPendingBit(ADC1, ADC_IT_EOC);
}
```

### Exercise 14.3 Voice Recorder

TBD

# Chapter 15

# NewLib

Throughout this book we've lived without the standard C libraries (`libc`) even though an implementation is distributed with the Sourcery tools that we use for building projects. The primary reason we have avoided `libc` thus far is to maintain clarity about what code is actually executing – as we'll see, introducing `libc` can quickly lead to obfuscation. A second reason is that `libc` is a memory hog – with only 8k SRAM available on the discovery board processor, there isn't much room to spare. However, the first consideration is now moot – you've had a pretty complete introduction to writing code for the STM32, and the second consideration is much less important if you move on to larger members of the STM32 family. There are many advantages to using `libc` including access to the standard library functions.

The implementation of `libc` distributed with the Sourcery tools and widely used for embedded systems is "newlib." newlib includes the `stdlib` functions (e.g. `abs`, `atoi`, `atoll`, `malloc`, and unicode support), `stdio` (`printf`, `sprintf`, `scanf`, ...), string support, and many others. newlib also includes an implementation of the standard math libraries `libm`. Unfortunately, the `stdio` functions are particular memory hogs as they allocate rather large memory buffers to support I/O. Furthermore, by default (for example, the Sourcery distribution) newlib is not configured to minimize memory use. Thus, we will also show how to compile newlib from sources to optimize memory use.

The architecture of newlib requires an external implementation of key system functions including – `open`, `close`, `read`, `write`, and `sbrk` – the last of these is the building block upon which `malloc` is built. In order to use `libc`, it is necessary to provide at least stubs for these and other functions. We will show how to design these stubs to support standard I/O using the STM32 USART

This chapter is organized as follows. We will begin with a simple example – "hello world" (surprise !) to illustrate the issues we must resolve in order to use newlib, and the memory issues that it raises. Recall that in Chapter 3 we pointed out that "hello world" is actually a complex program, although most of the complexity is "under the hood." Finally, we will show how to compile `libc` in a manner that minimizes the memory requirements (especially SRAM). If you're going to use `libc` on an STM32 variant with substantial SRAM (at least 20k) this step is unnecessary – you may use the libraries distributed with the Sourcery tools.

## 15.1 Hello World

"Hello World" is the most famous programming example in the C universe. As we pointed out in Chapter 3, it's actually pretty complex. Consider the program:

```
#include <stdio.h>
main() {
  printf("hello world\n");
}
```

`printf` is a library function which takes a format string and an optional list of parameters. It must stringify these parameters, merge them with the format string, and write the resulting string to `stdout`. What is `stdout` ? It's a buffered stream – the `stdio` libraries manage buffered streams. Data are buffered as they are written, and then the buffered data are written to a file or device. In this case, our intuition is that this stream should somehow be connected to a UART to print to a screen somewhere. While our use of `printf` is trivial, more complex invocations allocate memory in order to have space for performing any conversions.

If we attempt to compile "hello world" (create a project and try this !), we immediately learn that there are a number of undefined functions (Listing 15.1).

In a desktop environment, these all correspond to operating system calls – `libc` is just a code library, ultimately it needs access to an operating system API. In order to use newlib, we must provide the missing functionality in the form of procedures. Some of these we will replace with simple stubs, others (`read` and `write`) with code that accesses a uart.

As we mentioned, `libc` allocates memory – a lot of it – from the "heap." `libc` provides functions to manage the heap (`malloc` and `free`), but it de-

```
undefined reference to `_sbrk'
undefined reference to `_write'
undefined reference to `_close'
undefined reference to `_fstat'
undefined reference to `_isatty'
undefined reference to `_lseek'
undefined reference to `_read'
```

Listing 15.1: Undefined Functions in newlib

pends upon a system function `_sbrk` to allocate the memory that it manages. Consider Figure 15.1 repeated from Chapter 2 which illustrates the use of RAM by an executing program. The heap grows upward from the compiler allocated data towards the stack which grows downwards. Within the heap, memory is allocated by `malloc` and deallocated by `free`; however, whenever `malloc` has insufficient free space it asks for more memory via `_sbrk`, which has the effect of moving the heap end up (or down for a negative request). In a desktop system, this is implemented by allocated more memory pages, indeed `malloc` generally asks for memory blocks that are integral numbers of pages (e.g. 4096 bytes), which is a problem in a memory constrained system such as the discovery board.

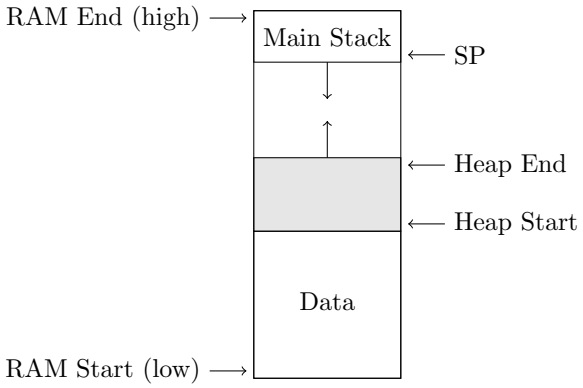



Figure 15.1: Program Memory Model

In order to implement `_sbrk` we need to know where the heap starts (i.e. the end of compiler/linker allocated memory) and what its physical limit is (i.e the maximum value for heap end). In the linker script, we define two values `_end` – the end of the data segment – and `_stackend` which is the limit

of the space reserved for the stack. [1]

Before providing an implementation of `_sbrk`, it is enlightening to read the Linux man page for `sbrk` (which calls the system function `_sbrk`).

> void *sbrk(intptr_t increment);
> ...
> brk() and sbrk() change the location of the program break, which defines the end of the process's data segment (i.e., the program break is the first location after the end of the uninitialized data segment). Increasing the program break has the effect of allocating memory to the process; decreasing the break deallocates memory.

A few other key points are in order. `_sbrk(0)` returns the current "program break." If the call cannot be satisfied, `_sbrk` returns -1 and sets the current `errno` to an appropriate error code. In `newlib`, `errno` is part of a structure used to make the library reentrant – this "reent" structure is another large source of memory usage which must be replicated in a multi-threaded environment. An implementation of `_sbrk` is shown in Listing 15.2

The remaining stubs are provided in Listing 15.3. The `read` and `write` operations utilize the `putc` and `getc` interface described in Chapter 5; in both cases we restrict our transfers to a single character. Notice that most of the remaining stubs simply return an error code; an exception is `_isatty` which returns 1 since we're using the uart as a terminal. `_fstat` which provides meta data for open files always sets the file mode to `S_IFCHR` which denotes a character oriented device.

There is one final detail required to use newlib. The startup code must call `_libc_init_array(void)` before main. This ensures that any required data structures are correctly initialized. In our startup code, we provide a "weak" default implementation for the case where we compile without `libc`.

```
void __attribute__ ((weak)) __libc_init_array (void){}

void Reset_Handler(void) {
   ...
 __libc_init_array();
 main();
   ...
}
```

```
#include <errno.h>

// defined in linker script

extern caddr_t _end, _stackend;

caddr_t _sbrk(int nbytes){
  static caddr_t heap_ptr = NULL;
  caddr_t        base;

  if (heap_ptr == NULL) {
    heap_ptr = (caddr_t)&_end;
  }

  if ((caddr_t) &_stackend > heap_ptr + nbytes) {
    base = heap_ptr;
    heap_ptr += nbytes;
    return (base);
  } else {
    errno = ENOMEM;
    return ((caddr_t)-1);
  }
}
```

Listing 15.2: `_sbrk` Implementation

### Exercise 15.1 Hello World

Complete and test the hello world example. You should put all the stub code in a single file – `syscalls.c` and compile this with your code.

[1] Calculating stack sizes can be challenging – especially in the presence of library code. It's best to relatively conservative !

```
int _close(int file) {
  errno = ENOTSUP;
  return -1;
}

int _fstat(int file, struct stat *st) {
  st->st_mode = S_IFCHR; // character device
  return 0;
}

int _isatty(int file) {
  return 1;
}

int _link(char *old, char *new) {
  errno = EMLINK;
  return -1;
}

int _lseek(int file, int ptr, int dir) {
  errno = ENOTSUP;
  return -1;
}

int _open(const char *name, int flags, int mode) {
  errno = ENOTSUP;
  return -1;
}

int _read(int file, char *ptr, int len) {
  if (len){
      *ptr = (char) uart_getc(USART1);
      return 1;
  }
  return 0;
}

int _unlink(char *name) {
  errno = ENOENT;
  return -1;
}

int _write(int file, char *ptr, int len) {
  if (len) {
    uart_putc(*ptr, USART1);
    return 1;
  }
  return 0;
}
```

Listing 15.3: Basic Syscall stubs

## 15.2 Building newlib

The distribution of newlib with the Sourcery tool was not compiled for minimum memory usage. You can build your own version by downloading the newlib sources and using the following build process:

```
mkdir newlib-build
cd newlib-build
export CFLAGS_FOR_TARGET=''-g -O2 -DSMALL_MEMORY''
/path_to_newlib_source/configure --target=arm-none-eabi
    ↪--prefix=/target-directory --disable-newlib-supplied-syscalls
    ↪--disable-libgloss --disable-nls
```

# Chapter 16

# Real-Time Operating Systems

The STM32 hardware is capable of simultaneously performing actions on all its various communication buses – for example reading audio files from an SD card on the SPI bus, playing these audio files through the DAC, monitoring Nunchuks over the I2C bus and logging messages through the UART. However, coordinating these parallel activities through software can be a real challenge – especially where hard timing constraints must be satisfied. One common strategy is to partition responsibility for multiple activities among separate threads – each of which acts autonomously – which are scheduled based upon priorities. For example, threads with tight timing constraints are given higher priorities than other threads.

Threads provide a way to partition the logic of a program into separate tasks. Each thread has its own state and appears to execute as an autonomous program while sharing data with other threads. In a uni-processor such as the STM32, threads are executed in an interleaved manner with access to the processor controlled by a scheduler. Whenever an interrupt occurs, there is an opportunity to suspend the current thread and resume a blocked thread. A timer interrupt provides the mechanism to "time-slice" the processor allowing each ready thread the opportunity to make forward progress.

Coordination of hardware tasks by multiple threads is enabled through synchronization objects. For example, a thread which is attempting to transmit a stream of data through a UART cannot make forward progress when the output buffer is full. In this case, the thread should "wait" allowing other threads to execute. Later when space is freed in the transmit buffer, for example by an interrupt handler, the waiting thread can be "signaled" to resume. This wait/signal pattern is implemented using a synchronization object called a "semaphore."

The decision to structure a program around threads should not be taken lightly because there are many potential pitfalls Threads require additional RAM memory because each thread requires a separate stack; in memory constrained devices such as the processor on the discovery board this can be a real problem. Threads can overflow their stacks if insufficient space is allocated – it can be difficult to accurately estimate the space required. Threads offer ample opportunity for subtle bugs in the form of race conditions wherever threads share data. Finally, threads are extremely difficult to debug. While one would like to trace the execution of a single thread, breakpoints are typically at the instruction level and, with shared code, halt any thread executing the instruction. Furthermore, GDB is not integrated with most thread packages and hence it is not easy to see the state of threads other than the currently halted one.

There are many real-time operating systems available for the STM32. In this chapter we use FreeRTOS because it is relatively simple and is available in source code form. FreeRTOS provides a basic kernel with a small set of synchronization primitives. In contrast, Chibios provides a complete hardware abstraction layer with drivers for many of the STM32 devices. FreeRTOS serves our pedagogical purposes better because it is easier to "look under the hood", but Chibios provides a significantly richer foundation for building large projects. The choice to utilize FreeRTOS has two negative consequences – key documents are available only by purchase, and the kernel requires dynamic memory allocation which is not desirable in memory constrained situations. In contrast, the Chibios kernel is statically allocated and all documents are freely available.

The remainder of this chapter is organized as follows. We begin with a discussion of threads, their implementation, and the FreeRTOS API for thread management. We then discuss key synchronization primitives and show how they can be used to control access to shared hardware, shared data (e.g. a FatFS file system) and coordinate scheduling with interrupt handlers (UART receive and send queues).

**Note** the stm32vl discovery board doesn't really have sufficient memory to support both libc (newlib) and FreeRTOS simultaneously. Thus, we do not address additional issues that arise when combining the two (the FreeRTOS distribution provides further information on this topic).

## 16.1 Threads

The single threaded programs which we have developed throughout this book organize RAM with all statically defined data allocated space in the low address portion of RAM and the program stack initialized at the top (highest address) in RAM. As the program executes, the stack grows downward on entry to procedures and shrinks upward on procedure exit. When interrupts occur, key portions of the execution context are pushed onto the main stack – see Chapter 11 for further explanation. In a multi-threaded environment, the main stack is used during initialization and then, primarily as an interrupt stack. Each active thread is allocated its own stack within RAM as illustrated in Figure 16.1.

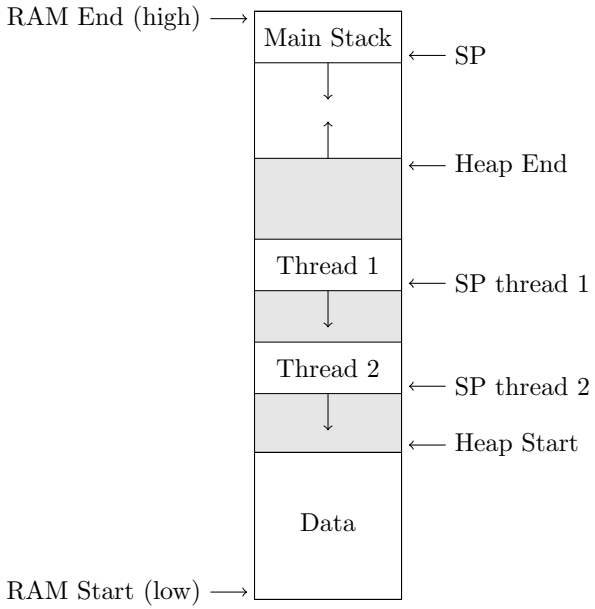



Figure 16.1: RAM Layout with Executing Threads

The data area is allocated statically at link time and includes all global and static variables both initialized and uninitialized. The area above the data is used for dynamic memory allocation (e.g. malloc). This area, called the heap, is expanded by the memory allocator as required. The main stack is placed at the high end of RAM by the linker and grows downward. In FreeRTOS, the thread stacks are allocated blocks within the heap. Space is

allocated within each stack by executing code (e.g. at procedure entry). Correct program execution requires that no stack ever grow beyond its allocated region. While there are runtime tests that can help detect such an unhappy event, it is difficult to guarantee that an overflow will never occur.

FreeRTOS supports threads (called "tasks" in FreeRTOS) with a prioritized pre-emptive kernel – at any moment, the thread with the highest priority is allowed to run. Threads with identical priorities are "time-sliced" with each being allowed to run for a fixed period before being pre-empted. To understand the key ideas of the kernel, consider the thread states in Figure 16.1. Every thread (task) is in one of four states – Ready, Running, Blocked, or suspended. When a thread is created (0) it is put into the Ready state. The (single) running thread is in the Running state. A running thread may be pre-empted and returned to the Ready state (1) in which case a ready thread is moved to the Running state (2). A running thread may also be blocked (3) by calling a blocking API function (such as waiting on a semaphore). A blocked thread may be made ready (4) when un-blocked by the actions of another thread or interrupt handler. FreeRTOS has an additional Suspended state which we ignore for now.

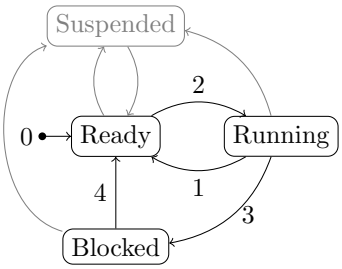



Figure 16.2: Thread (Task) States in FreeRTOS

The code for a thread (task) is defined by a C function that never returns as in:

```
void threadFunction(void *params){
  while(1) {
    // do something
  }
}
```

When "created" a thread is passed a pointer to a parameter structure – many threads may share a single function, but can be specialized through

the parameters. A thread is created with a function, a name, a stack size, optional parameter pointer, a priority, and (optionally) a location to store a unique "handle" to the thread.

```
portBASE_TYPE xTaskCreate(
       pdTASK_CODE pvTaskCode,
       const portCHAR * const pcName,
       unsigned portSHORT usStackDepth,
       void *pvParameters,
       unsigned portBASE_TYPE uxPriority,
       xTaskHandle *pvCreatedTask
);
```

The FreeRTOS kernel allocates memory for the stack and creates the necessary thread data structure. [1]

The stack size is a key parameter – too small and the thread will overflow its stack and corrupt memory, too large and memory is wasted. Since FreeRTOS uses the stack to hold the context for non-running tasks (68 bytes for 17 registers !) the minimum stack size is actually relatively large. Furthermore, if the task code calls any library functions, the stack will need to be big enough to hold any data allocated on the stack and any registers saved. Most threads will need 128-256 bytes.

The FreeRTOS kernel also creates an "idle thread" which has the lowest priority and runs whenever no other thread can run. (The idle thread has a stack too !). The basic flow of a multi-threaded program follows:

```
main() {

  // include misc.h

  NVIC_PriorityGroupConfig( NVIC_PriorityGroup_4 );

  // Initialize hardware
      ...
  // Create tasks

  xTaskCreate( ... );
      ...
  // Start Scheduler

  vTaskStartScheduler();
}
```

[1] FreeRTOS requires a memory allocator to dynamically allocate key data structures.

As with all the programs considered this far, code begins with hardware initialization. One hardware initialization that we call out is the NVIC configuration. FreeRTOS requires preemptive thread priorities; here we enable the NVIC to support 16 priorities (and no sub-priorities). This is the configuration assumed by the STM32 port of FreeRTOS. Once hardware has been initialized, initial set of threads are created. While FreeRTOS allows dynamic thread allocation, the limited RAM (8K) available on the Discovery board virtually requires that we create all threads initially. The scheduler never returns (there is an API call to exit the scheduler, but that isn't particularly useful for our applications). A more complete example, with two threads that blink two LEDs is illustrated in Listing 16.1. Notice that the threads utilize `vTaskDelay` to sleep.

```
static void Thread1(void *arg) {
  int dir = 0;
  while (1) {
    vTaskDelay(300/portTICK_RATE_MS);
    GPIO_WriteBit(GPIOC, GPIO_Pin_9, dir ? Bit_SET : Bit_RESET);
    dir =  1 - dir;
  }
}

static void Thread2(void *arg) {
  int dir = 0;
  while (1) {
    vTaskDelay(500/portTICK_RATE_MS);
    GPIO_WriteBit(GPIOC, GPIO_Pin_8, dir ? Bit_SET : Bit_RESET);
    dir =  1 - dir;
  }
}

int main(void)
{
  // set up interrupt priorities for FreeRTOS !!

  NVIC_PriorityGroupConfig( NVIC_PriorityGroup_4 );

  // initialize hardware

  init_hardware();

  // Create tasks

  xTaskCreate(Thread1,                    // Function to execute
        "Thread 1",                 // Name
        128,                        // Stack size
        NULL,                       // Parameter (none)
              tskIDLE_PRIORITY + 1 ,     // Scheduling priority
        NULL                        // Storage for handle (none)
        );
  xTaskCreate(Thread2, "Thread 2",  128,
        NULL, tskIDLE_PRIORITY + 1 , NULL);

  // Start scheduler

   vTaskStartScheduler();

  // Schedule never ends

}
```

Listing 16.1: FreeRTOS Example

## 16.2 FreeRTOS Configuration

```
FreeRTOS/
├── Demo/CORTEX_STM32F100_Atollic/Simple_Demo_Source/
│   └── FreeRTOSConfig.h
└── Source/
    ├── include/
    ├── list.c
    ├── portable/
    │   ├── MemMang/
    │   │   └── heap_1.c
    │   └── GCC/ARM_CM3/
    │       ├── port.c
    │       └── portmacro.h
    ├── queue.c
    ├── tasks.c
    └── timers.c
```

Figure 16.3: Key Parts of FreeRTOS Distribution

The FreeRTOS source code is freely available from `http://sourceforge.net/projects/freertos/files/`. The distribution can be a little overwhelming. The key files which we use are illustrated in Figure 16.3. In creating a sample application, it will be necessary to augment your makefile to include the necessary paths, as illustrated in Listing 16.2.

```
FreeRTOS = ... path_to_FreeRTOS ...
CFLAGS  +=  -I$(FreeRTOS)/include -DGCC_ARMCM3
...
vpath %.c $(FreeRTOS)/
vpath %.c $(FreeRTOS)/portable/MemMang
vpath %.c $(FreeRTOS)/portable/GCC/ARM_CM3
...
OBJS+=   tasks.o queue.o list.o timers.o heap_1.o port.o
```

Listing 16.2: Build Paths for FreeRTOS

Every FreeRTOS project requires a configuration file. This is used to set key parameters of the project; for example, enabling or disabling kernel features – `FreeRTOSConfig.h`. We based our projects on the configuration file shown in Figure 16.3. The configuration file serves several purposes. First, it

defines key parameters – for example clock rate, heap size, number of priorities, etc. Second, it defines features to be enabled in a given build – features are enabled (1) or disabled (0) by defines of the following form:

```
#define INCLUDE_xxx 0+
```

Examples of key features include

```
#define configUSE_MUTEXES                1
#define configCHECK_FOR_STACK_OVERFLOW   1
#define configUSE_RECURSIVE_MUTEXES      0
#define configUSE_COUNTING_SEMAPHORES    0
```

Finally, FreeRTOS also provides the opportunity for user code to "hook" into the kernal at key places:

```
#define configUSE_MALLOC_FAILED_HOOK     0
#define configUSE_IDLE_HOOK              0
#define configUSE_TICK_HOOK              0
```

If you compile a project with FreeRTOS and have linker errors for missing functions then you should check to see if a corresponding hook was defined. You then have the option to either provide the required function or disable the hook.

### Exercise 16.1 RTOS – Blinking Lights

Complete the blinking lights demo. You should disable any features which your project does not need by making a local copy of the configuration file. You will find it necessary to reduce the heap size (from 7k to 5k) in order for your project to build.

## 16.3 Synchronization

Threads cannot safely share either data structures or hardware without some synchronization mechanisms. Consider the following `getchar` implementation discussed in Chapter 9:

```
int getchar(void){
  while (USART_GetFlagStatus(USART1, USART_FLAG_RXNE) == RESET);
  return USARTx->DR & 0xff;
}
```

Recall that this works by reading the UART status register until the receive data resister is "not empty" and then reading the register. Suppose that two threads access the UART – if thread 1 reads the status register and finds the data register "not empty", but is then preempted by thread 2 which also tests the status and reads the data register, then when thread 1 is resumed, its knowlege of the status register will be incorrect and it may read garbage from the data register.

The key idea to making this work is finding a method to ensure exclusive access by thread 1 to the UART receive register. One approach is to prevent pre-emption by disablng interrupts; however, consider what would happen if no character arrived or simply arrived after a substantial delay. What is needed is a mechanism to block only those threads which are competing for access to the UART receive hardware. The solution is to use a semphore:

```
xSemaphoreHandle uartRcvMutex;

uart_init(...){
  ...
  uartRcvMutex = xSemaphoreCreateMutex();
}

int getchar(void){
  int c;
  if (xSemaphoreTake(uartRcvMutex, portMAX_DELAY) == pdTRUE)
  {
    while (USART_GetFlagStatus(USART1, USART_FLAG_RXNE) == RESET);
    c = USARTx->DR & 0xff;
    xSemaphoreGive(uartRcvMutex);
  }
  return c;
}
```

A semaphore is a standard sychronization primitive (indeed the first one described for structuring operating system code. The idea is provide a thread safe interface to manage shared resources. A thread requesting a resource is either granted the resource or blocked by the scheduler until the resource becomes available. A thread releasing a resource may, as a side effect, un-block a waiting thread.

FreeRTOS supports three types of semaphores – binary semaphores, mutexes, and counting semaphores. Mutexes and binary semaphores are similar, but behave differently with respect to thread priority. Here we use a Mutex. There are two operations of note – *take* and *give*. When the mutex is initialized it has a single "token". *Take* removes the token if available, or

blocks the calling thread and places it on a list associated with the Mutex if no token is available. *Give* restores the token. The major difference between mutexes (binary semaphores) and counting semephores is that the latter may have multiple tokens.

Another Mutex can be added to similarly protect `putchar`. Protecting `getchar` and `putchar` with Mutexes will prevent data races such as the one above; however, this may not yield the expected behavior. Suppose multiple threads call `putchar` through a procedure `putstring`:

```
void putstring(char *s){
  while (s && *s)
    putchar(*s++);
}
```

In this case, the output of two threads simultaneously writing strings may be interleaved !

### Exercise 16.2 Multiple Threads

Write a program with two threads continuously printing strings to UART1. A third thread should blink one of the leds at 2Hz. You should use thread parameters to specialize a common thread function for the two printing threads. Try your code first using just a Mutex on `putchar`. Then come up with a solution that prevents interleaving of strings.

## 16.4 Interrupt Handlers

The `getchar` code above has a major flaw – the thread holding the mutex will continue to spin on the flag test until it succeeds. Ideally, we need a mechanism to allow the waiting thread to sleep until space is available. With peripherals such as UARTs we have already seen that interrupt code can handle the actual changes in hardware status. In Section 11.5 we showed how this might work. The remaining issue is how to communicate these hardware events with interrupt handlers. As we showed previously, we would like to associate a pair of queues with the transmit and receive interfaces of the UART. Where previously our `putchar` code failed when the trasmit queue was empty and our `getchar` code failed when the receive queue was empty, in a multi-threaded environment we would like to block a thread in either of these cases and have the interrupt handler wake the thread.

FreeRTOS provides its own blocking queue primitives (indeed, semaphores and mutexes are special cases of queues):

```
xQueueHandle xQueueCreate(
                    unsigned portBASE_TYPE uxQueueLength,
                    unsigned portBASE_TYPE uxItemSize);
```

A queue is created with both a length and data size. Items are queued by copying them rather than by reference (one can always queue a pointer, but think carefully before doing this !). The two interface procedures that we require are "send" and "receive". Notice that each has a "timeout" which may range from 0 (don't wait if the operation would fail to `portMAX_DELAY` to wait forever.

```
portBASE_TYPE xQueueSend( xQueueHandle xQueue,
                    const void * pvItemToQueue,
                    portTickType xTicksToWait );

portBASE_TYPE xQueueReceive(xQueueHandle xQueue,
                    void *pvBuffer,
                    portTickType xTicksToWait );
```

Both of these interfaces may be used by threads, but not interrupt handlers, which must use special versions of these:

```
portBASE_TYPE xQueueSendFromISR( xQueueHandle pxQueue,
                 const void *pvItemToQueue,
                 portBASE_TYPE *pxHigherPriorityTaskWoken);

portBASE_TYPE xQueueReceiveFromISR( xQueueHandle pxQueue,
                void *pvBuffer,
                portBASE_TYPE *pxTaskWoken);
```

Notice, these "ISR" (interrupt service routine) interfaces have a different third parameter – a flag is returned if a task has been "woken" by the call. In this situation, the interrupt handler should notify the kernel as we shall demonstrate in a moment.

**Note:** It is very important that interrupt handlers accessing the FreeRTOS API have lower priorities (higher numbers) than the value `LIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY` defined in the FreeRTOS configuration file.

We now build an interrupt driven version of the uart as shown in Listings 16.3 and 16.4

### Exercise 16.3 Multithreaded Queues

```
int uart_putc(int c){
  xQueueSend(UART1_TXq, &c, portMAX_DELAY);
  // kick the transmitter interrupt
  USART_ITConfig(USART1, USART_IT_TXE, ENABLE);
  return 0;
}

int uart_getc (){
  int8_t buf;
  xQueueReceive(UART1_RXq, &buf, portMAX_DELAY);
  return buf;
}
```

Listing 16.3: Interrupt Driven UART

Complete an interrupt driven UART with flow control using queues along with a multi-threaded program that exercises both send and receive.

```
void USART1_IRQHandler(void)
{
  portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;

  if(USART_GetITStatus(USART1, USART_IT_RXNE) != RESET){

      uint8_t  data;

      USART_ClearITPendingBit(USART1, USART_IT_RXNE);

      data = USART_ReceiveData(USART1) & 0xff;
      if (xQueueSendFromISR(UART1_RXq, &data,
          ↪&xHigherPriorityTaskWoken) != pdTRUE)
        RxOverflow = 1;
  }

  if(USART_GetITStatus(USART1, USART_IT_TXE) != RESET) {
      uint8_t data;

      if (xQueueReceiveFromISR(UART1_TXq, &data,
          ↪&xHigherPriorityTaskWoken) == pdTRUE){
          USART_SendData(USART1, data);
        }
      else  {

        // turn off interrupt

        USART_ITConfig(USART1, USART_IT_TXE, DISABLE);
        }
    }
    // Cause a scheduling operation if necessary

    portEND_SWITCHING_ISR( xHigherPriorityTaskWoken );
}
```

Listing 16.4: Interrupt Driven UART (Handler)

## 16.5 SPI

The other communication interfaces we have introduced in this book also need to be reconsidered in the face of multi-threading. The I2C interface is much like the Uart interface in that a semaphore can be added directly to the interface code to protect the read/write interfaces (only one semaphore is needed). SPI is somewhat different – whereas the I2C protocol includes device addressing in the transmitted data, SPI uses separate select lines for

each device. Our use of the SPI interface looks something like:

```
select();
spi_operation();
deselect();
```

where select and deselect are device specific. If we followed the UART pattern of accessing a single semaphore inside the SPI operations then it would be possible have one thread interfere with another. We have a couple of design options. We could simply make the semaphore for the SPI interface into a global object an require all users of the interface to first request the sempahore. This has the significant disadvantage of requiring application code writers to be too familar with the low-level synchronization assumptions for the interface and runs a significant risk of leading to flawed code. We could add the select GPIO pin as a (pair) of parameters to the SPI interface as in:

```
void spiReadWrite(SPI_TypeDef* SPIx, uint8_t *buf, uint8_t * buf,
    ↪int cnt,
  uint16_t pin, GPIO_TypeDef* GPIOx);
```

Or, we could pass a callback function in which the SPI interface could use to select/deselect the device after taking the semaphore

```
typedef void selectCB_t(int sel);
void spiReadWrite(SPI_TypeDef* SPIx, uint8_t *buf, uint8_t *buf,
    ↪int cnt,
  selectCB_t selectCB);
```

This final approach seems preferable as it doesn't burden the SPI interface with detailed knowledge of how the select/deselect operates.

The SPI interface is also used for rather long data transfers utilizing DMA. For example, we use it to transfer blocks of data for the FatFS. In a multi-threaded environment, it is desirable that DMA proceed in the background. In our current implementation we use busy waiting for DMA to complete

```
while (DMA_GetFlagStatus(dmaflag) == RESET) { ; }
```

As with the UART code, this busy waiting can be replaced by blocking the thread initiating the DMA operation on a synchronization object (in this case a binary semaphore is appropriate) and utilizing an interrupt handler to unblock the thread (as illustrated below).

```
static void  vExampleInterruptHandler( void )
{
  portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;

  // handle interrupt

    ...

  // release semaphore

  xSemaphoreGiveFromISR( xBinarySemaphore,
      ↪&xHigherPriorityTaskWoken);
  portEND_SWITCHING_ISR( xHigherPriorityTaskWoken );
}
```

In Chapter 13 we showed how to create DMA interrupt handlers. For the SPI interface we need to enable interrupts for the "transfer complete" case.

### Exercise 16.4 Multithreaded SPI

Create a SPI interface supporting multi-threading and DMA. You should use callbacks for device selection. Test your interface with the LCD with two threads competing to use the interface for drawing colored rectangles at random locations (different colors for the two threads).

## 16.6 FatFS

The FatFS code has some builtin support for multi-threading which must be enabled in the `ffconf.h`.

```
#define _FS_REENTRANT 0      /* 0:Disable or 1:Enable             */
#define _FS_TIMEOUT   1000   /* Timeout period in unit of ticks   */
#define _SYNC_t       HANDLE /* O/S dependent type of sync object */
```

It is up to the user to define an appropriate `_SYNC_t` type and to provide implementations for `ff_req_grant`, `ff_rel_grant`, `ff_del_syncobj` and `ff_cre_syncobj`. It is also necessary to modify the low-level device interface to utilize a modified SPI interface and to replace busy waiting with appropriate RTOS delay functions.

### Exercise 16.5 Multithreaded FatFS

Rewrite the low-level interface for the FatFS to use the new SPI interface and to remove busy waiting – your changes should work either in a single

threaded or multi-threaded environment (use the `_FS_REENTRANT` flag appropriately !. Test your code with a multi-threaded program that plays audio files and displays images from an SD card. One thread should play an audio file while another displays an image.

## 16.7 FreeRTOS API

The FreeRTOS API is documented at `www.freertos.org`. Here we identify the key API functions used throughout this chapter.

| | |
|---|---|
| **Task Creation** | |
| xTaskCreate() | Create task |
| xTaskDelete() | Delete task |
| **Task Utilities** | |
| vTaskDelay() | Delay from now |
| vTaskDelayUntil() | Delay from previous wake time |
| **Kernel Utilities** | |
| taskYIELD() | Force a context switch |
| vTaskSuspendAll() | Prevent current task from being preempted |
| vTaskResumeAll() | Resume normal scheduling |
| **Queues** | |
| uxQueueMessagesWaiting() | Number of messages in queue |
| xQueueCreate() | Create queue |
| xQueueDelete() | Create queue |
| xQueueSend() | Send to queue |
| xQueueReceive() | Receive from queue |
| xQueueSendFromISR() | ISR send to queue |
| xQueueReceiveFromISR() | ISR Receive from queue |
| **Sempahore** | |
| vSemephoreCreateBinary() | Create a binary semaphore |
| vSemephoreCreateCounting() | Create a counting semaphore |
| vSemephoreCreateMutex() | Create a mutex |
| vSemephoreCreateTake() | Take from semaphore |
| vSemephoreCreateGive() | Give to semaphore |
| vSemephoreCreateGiveFromISR() | Give from ISR to semaphore |

Table 16.1: FreeRTOS API – Key Calls

## 16.8 Discusion

Developing a project with FreeRTOS is a lot of work ! All of the device interfaces have to be written to support multi-threading and great care taken to ensure freedom from race conditions. An alterantive is to use an RTOS with a hardware abstraction layer such as Chibios where much of the heavy lifting has been done for you.

You probably have also found that debugging multi-threaded code is very challenging. The message is clear – use threads only when really needed.

# Chapter 17

# Next Steps

In the preceding chapters you've learned many of the basic skills needed to build interesting projects using STM32 processors and off-the-shelf modules. In this chapter, I provide pointers to many additional modules and guidance for how the skills you've learned can be applied to develop code to interface to these. By now, it should be evident that SRAM is a major limitation of the STM32 VL Discovery Processor. Both newlib and FreeRTOS need much more memory to operate and large projects can greatly benefit from the additional abstraction layer provided by these libraries. Thus, I begin with a discussion of boards providing STM32 processors with larger memories. It should also be evident by now that using a SPI-based LCD, while adequate for basic display isn't really up to the performance needs of complex graphics; higher performance is achieved at the expense of a new interface.

The interfaces you've learned can be applied to a wide variety of new sensors including climate (temperature/pressure/humidity), position and inertial (accelerometer/gyroscope/magnetometer/GPS), force (flex sensors) – I'll discuss some of these, but a useful step is to peruse the offerings of companies such as Sparkfun while noting the interfaces their modules require. The basic serial interface enables the use of various wireless communication devices including bluetooth, wifi, and GSM (cell phone) while the SPI interface enables the use of low-power radio devices. Finally timers provide the key to motion control including stepper motor and DC control with position feedback.

This chapter is intended primarily to illustrate the world your new skills enable – not as definitive guide to interfacing and using new modules; that job is up to you.

## 17.1 Processors

The STM32 F100 micro-controller used on the VL Discovery board is one member of a series of STM32 F1xx components which includes:

- Value line STM32 F100 – 24 MHz CPU with motor control and CEC functions
- Access line STM32 F101 – 36 MHz CPU, upt to 1 MByte Flash
- USB Access line STM32 F102 – 48 MHz with USB file system
- Performance line STM32 F103 – 72 MHz, up to 1 Mbyte Flash with motor control, USB and CAN
- Connectivity line STM32 F105/STM32 F107 – 72 MHz CPU with Ethernet MAC, CAN and USB 2.0 OTG

All of these lines of components utilize the same standard peripheral libraries. They do require modifications to the linker script (particularly the definitions of memory regions) and startup code (the definitions of exception vectors); however neither task is difficult. They also include peripherals that you have not seen including support for external memory (FSMC), a higher level SD interface (SDIO) and communication interfaces such as USB, Ethernet, and CAN. The communication interfaces generally require significant additional software libraries and can be very difficult (e.g. USB) to debug.

All of the examples from this book should work without modification on any processor from the the STM32F1xx families. A few changes will be required to the files in the template directory provided with this book. The linker script from STM32-Template needs the following modification:

```
MEMORY
{
  RAM (rwx)  : ORIGIN = 0x20000000, LENGTH = 8K
  FLASH (rx) : ORIGIN = 0x08000000, LENGTH = 128K
}
```

[frame=none]

The lengths of the two memory regions should be modified to reflect the component being use. `Makefile.common` needs to have its `LDSCRIPT, PTYPE, STARTUP` variables modified as appropriate. Finally, the vector table in `startup_stm32f10x.c` will need to be updated to reflect the vectors in the

part family being used. The standard peripheral library includes assembly language startup files that may serve as a guide. Note that the vector table ends with a "magic" number that is necessary if you build projects to boot out of SRAM.

There are additional STM32 families including STM32 F0 (Cortex-M0 based), STM32 L1 (Ultra low-power Cortex-M3), STM32 F2 (high performance Cortex-M3 – 120MHz), and STM32 F4/F3 (Cortex-M4 based). Each of these requires using a different standard peripheral library. Although many of the peripherals are the same or enhancements of those in the STM32 F1 series, some care will be required to port the code in this document to these devices. There are extremely attractive "discovery boards" for the STM32 L1, STM32 F0, and STM32 F4 parts which are worth investigating. At this time, the gdb interface code used in this document supports all but the STM32 F0 device; however, it appears that openocd `openocd.sourceforge.net` can support them all.

For the reasons described above, the simplest upgrade from the STM32F100 is another member of the STM32F1xx family. Sparkfun sells a header board for the STM32F103 with 20K bytes of RAM and 128K bytes of flash for approximately $40. – the same board is also carried by Mouser. This board, produced by Olimex, also has a USB connector as the STM32F103 has a USB slave peripheral (not for the faint of heart !). Several similar development boards are available from various ebay vendors for $15-$25. For roughly $30 it is possible to buy an STM32F103 board with a 2.8" TFT touch-screen on ebay. There are excellent value and support high performance graphics through the FSMC parallel interface peripheral of the STM32F103.

Most of the readily available STM32 boards provide a standard JTAG connector for the debugger interface. While it is feasible to use the STM32VL Discovery board to communicate with this interface, the wiring issues may become annoying. ST sells a separate ST-LINK/V2 debugger module which interfaces directly to the JTAG connector. This module is available from both Mouser (`www.mouser.com`) and Digikey (`www.digikey.com`) for approximately $22 and would be a good investment for working with STM32 boards. The ST-link/V2 protocol is supported by the gdb driver used in this document – indeed it cause fewer issues in an OS X or Linux environment that the V1 protocol.

As mentioned above, more powerful displays generally require using a parallel interface such as the FSMC provided by the STM32 F103. Taking advantage of the enhanced capabilities requires a more sophisticated graphics

library. There are a number of open-source and commercial libraries available. ST provide an STM32 library [22]. While the STM32 library includes a hardware abstraction layer (HAL) which can be modified to support various devices including LCDs, touch-screens, and joysticks.

Recently ST Microelectronics has released discovery boards with STM32F4 and STM32F3 micro-controllers. These are amazing value. For example, the STM32F3 Discovery board includes a 9 degree inertial motion unit (accelerometer, gyroscope, compass) for less than $15. The biggest barrier to entry will be the need to adapt to a new standard peripheral library. While I am confident that the examples from this book will port with modest work, some peripherals may have changed considerably – for example the initialization of the GPIO pins has additional parameters.

## 17.2 Sensors

There are many sensors available that have SPI, I2C, or analog interfaces. In this section we discuss a few of these. Interested readers should peruse the Sparkfun site for ideas.

### Position/Inertial Measurement

ST, Analog devices, and Freescale all make accelerometer devices – many of these are supported by break-out boards from Sparkfun.com which provides a handy buying guide `http://www.sparkfun.com/tutorials/167`. In general, these devices use I2C, SPI, or analog interfaces. A complete "inertial measurement unit" generally contains a 3-axis accelerometer, 3-axis gyroscope, and 3-axis magnetometer; ST produces a module containing all three of these devices in a DIP form-factor (STEVAL-MKI108V2) which is available from Mouser for $29.

Another useful position sensing device is a GPS receiver. Sparkfun sells a nice unit based upon the Venus638FLPx for $50. Like most such units, this communicates with asynchronous serial – simply hook it up to a UART.

### Environmental Sensors

Environmental sensors include temperature, humidity, and pressure. These are widely available with I2C interfaces, although some such as Maxim/Dallas 1-wire devices require a special protocol. The 1-wire protocol is not directly supported by the STM32 peripherals, but can be implemented with

timers and GPIO; the Saleae Logic includes a protocol analyzer for 1-wire devices. Note that there are several similar, but incompatible 1-wire style buses; however, most are relatively simple.

In addition to digital protocols, environmental sensors frequently produce an analog signal. Examples include the various gas sensors (alcohol, carbon monixide, hydrogen,...) sold by Sparkfun and others.

### Motion and Force Sensors

There are many force/position sensors whose resistance varies with input. Measuring these requires a reference voltage and resistance to translate the sensor resistance to an analog voltage.

### ID – Barcode/RFID

Sparkfun and others sell RFID and barcode readers with serial and I2C interfaces.

### Proximity

Proximity sensors include both ultrasonic and infrared ranging modules. We have demonstrated the use of ultrasonic sensors in Chapter 10. Reflective IR sensors frequently require a digital output to power an LED and an analog input to measure the reflected light intensity.

## 17.3 Communication

Many interesting projects require enhanced communication either wireless or wired. In the wireless domain, inexpensive modules bluetooth, wifi, and even GSM cellular phones are available with serial interfaces. For low power applications, dedicated radio links such as the Nordic nRF24L01 utilize SPI interfaces. Finally, it is relatively easy to interface to standard infrared remote controls with a simple IR detector.

## 17.4 Discussion

I hope the ideas discussed in this chapter give you some idea of the possibilities. There is a vibrant maker community that can provide practical guidance – you might seek out examples developed for the arduino platform.

I believe the techniques presented in this book will give you a firm foundation for experimenting

# Attributions

Figure 1.10 was obtained from http://commons.wikimedia.org/wiki/File:Potentiometer.jpg under the Creative Commons Attribution-Share Alike 3.0 Unported license (http://creativecommons.org/licenses/by-sa/3.0/deed.en). This image is also used in Figure 14.1

Figure 1.6 is covered by the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported license. The image is due to www.sparkfun.com.

# Bibliography


[1] ARM Limited. Cortex-M3 technical reference manual, 2006.

[2] ARM Limited. Procedure call standard for the ARM® architecture, October 2009. IHI 0042D.

[3] ChaN. FatFs generic FAT file system module, 2011. http://elm-chan.org/fsw/ff/00index_e.html. Accessed April 2012.

[4] F. Foust. Secure digital card interface for the MSP430, 2004. http://alumni.cs.ucr.edu/~amitra/sdcard/Additional/sdcard_appnote_foust.pdf. Accessed July 2012.

[5] J. Jiu. *The Definitive guide to the ARM Cortext-M3*. Newnes, 2010.

[6] Microchip. 25AA160A/25LC160A 16K SPI bus serial EEPROM, 2004.

[7] NXP. ı2c bus specification and user manual (rev 03), June 2007. UM10204.

[8] C. Philips. Read wii nunchuck data into arduino. http://www.windmeadow.com/node/42. Accessed February 4, 2012.

[9] SD specifications, part 1, physcial layer simplified specification, version 3.01, May 2010.

[10] F. Semiconductor. Tilt sensing using linear accelerometers, 2012.

[11] Sitronix Technology Corporation. Sitronix st7735r 262k color single-chip TFT controller/driver, 2009.

[12] STMicroelectronics. How to get the best ADC accuracy in the STM32F10xxx devices, November 2008.

[13] STMicroelectronics. STM32F10xxx ı2c optimized examples, 2010.

[14] STMicroelectronics. User manual STM32 value line discovery, 2010. UM0919.

[15] STMicroelectronics. Low & medium-density value line, advanced ARM-based 32-bit MCU with 16 to 128 kb flash, 12 timers, ADC, DAC & 8 comm interfaces, 2011. Doc ID 16455.

[16] STMicroelectronics. Migrating a microcontroller application from STMF1 to STM32F2 series, 2011. AN3427: Doc ID 019001.

[17] STMicroelectronics. Migrating a microcontroller application from STMF1 to STM32L1 series, 2011. AN3422: Doc ID 018976.

[18] STMicroelectronics. Migration and compatibility guideslines for STM32 microcontroller applications, 2011. AN3364: Doc ID 018608.

[19] STMicroelectronics. Programming manual: Stm32f10xxx/ 20xxx/ 21xxx/ l1xxxx cortex-m3 programming manual, March 2011. PM0056.

[20] STMicroelectronics. Reference manual stm32f100xx advanced ARM-based 32-bit MCUs rev. 4, 2011. RM0041.

[21] STMicroelectronics. Reference manual stm32f101xx stm32f102xx stm32f103xx stm32f105xx and stm32f107xx advanced ARM-based 32-bit MCUs rev. 14, 2011.

[22] STMicroelectronics. STM32 embedded graphic objects/touchscreen library, 2011.

[23] wiibrew. Wiimote/Extension Controllers. `http://wiibrew.org/wiki/Wiimote/Extension_Controllers`. Accessed February 6, 2012.

[24] wikipedia. Wii remote. `http://en.wikipedia.org/wiki/Wii_Remote`. Accessed February 2, 2012.