RP2040 Assembly Language Programming

ARM Cortex-M0+ on the Raspberry Pi Pico

Stephen Smith

Apress®

RP2040 Assembly Language Programming: ARM Cortex-M0+ on the Raspberry Pi Pico

Stephen Smith
Gibsons, BC, Canada

ISBN-13 (pbk): 978-1-4842-7752-2 ISBN-13 (electronic): 978-1-4842-7753-9
https://doi.org/10.1007/978-1-4842-7753-9

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Aaron Black
Development Editor: James Markham
Coordinating Editor: Jessica Vakili

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-7752-2. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

This book is dedicated to my beloved wife and editor Cathalynn Labonté-Smith.

Table of Contents

About the Author

Stephen Smith is also the author of the Apress titles *Raspberry Pi Assembly Language Programming and Programming with 64-Bit ARM Assembly Language*. He is a retired software architect, located in Gibsons, BC, Canada. He's been developing software since high school, or way too many years to record. He was the chief architect for the Sage 300 line of accounting products for 23 years. Since retiring, he has pursued artificial intelligence, earned his Advanced HAM Radio License, and enjoys mountain biking, hiking, and nature photography, and is a member of the Sunshine Coast Search and Rescue group. He continues to write his popular technology blog at http://smist08.wordpress.com and has written two science fiction novels in a series, Influence and Unification, available on http://amazon.com.

About the Technical Reviewer

Stewart Watkiss is a keen maker, programmer, and author of *Learn Electronics with Raspberry Pi*. He studied at the University of Hull, where he earned a master's degree in electronic engineering, and more recently at Georgia Institute of Technology, where he earned a master's degree in computer science.

Stewart also volunteers as a STEM ambassador, helping teach programming and physical computing to schoolchildren and at Raspberry Pi events. He has created a number of resources using Pygame Zero, which he makes available on his website (www.penguintutor.com).

Acknowledgments

No book is ever written in isolation. I want to especially thank my wife, Cathalynn Labonté-Smith, for her support, encouragement, and expert editing.

I want to thank all the good folk at Apress who made the whole process easy and enjoyable. A special shout-out to Jessica Vakili, my coordinating editor, who kept the whole project moving quickly and smoothly. Thanks to Aaron Black, senior editor, who recruited me and got the project started. Thanks to Stewart Watkiss, my technical reviewer, who helped make this a far better book.

Introduction

There is an explosion of DIY electronics projects, largely fueled by the Arduino-based microcontrollers and Raspberry Pi computers. Electronics projects have never been easier to build, with hundreds of inexpensive modular components to choose from. People are designing robots, home monitoring and security systems, game devices, musical instruments, audio systems, and a lot more. The Raspberry Pi Pico is the Raspberry Pi Foundation's entry into the Arduino-style microcontroller market. A regular Raspberry Pi computer runs Linux and typically costs from $35 to $100 depending on memory and accessories. The Raspberry Pi Pico costs $4 and doesn't run an operating system.

To power the Raspberry Pi Pico, the Raspberry Pi Foundation designed a custom system on a chip (SoC), called the RP2040, containing dual ARM Cortex-M0+ CPUs along with a raft of device controller components. This combination of a powerful CPU and ease of integration has made this a great choice for any DIY project. Further, Raspberry sells the RP2040 chips separately, and other companies such as Seeed Studio, Adafruit, and Pimoroni are selling their own versions of this microcontroller with extra built-in features like Bluetooth or Wi-Fi. You can even buy RP2040 chips yourself for $1 each and build your own board.

At the basic level, how are these microcontrollers programmed? What provides the magical foundation for all the great projects that people build on them? Raspberry provides an SDK for C programmers as well as support for programming in MicroPython. This book answers these questions and delves into how these are programmed at the bare metal level and provides insight into the RP2040's architecture.

Assembly Language is the native, lowest-level way to program a computer. Each processing chip has its own Assembly Language. This book covers programming the ARM Cortex-M0+ 32-bit processor. To learn how a computer works, learning Assembly language is a great way to get into the nitty-gritty details. The popularity and low cost of microcontrollers like the Raspberry Pi Pico provide ideal platforms to learn advanced concepts in computing.

Even though all these devices are low powered and compact, they're still sophisticated computers with a multicore processor, programmable I/O processors, and integrated hardware controllers. Anything learned about these devices is directly relevant to any gadget with an ARM processor, which by volume is the number one processor on the market today.

In this book, we cover how to program ARM Cortex-M0+ processors at the lowest level, operating as close to the hardware as possible. You will learn the following:

- How to format instructions and combine them into programs, as well as details of the operative binary data formats
- How to program the built-in programmable I/O, division, and interpolation coprocessors
- How to control the integrated hardware devices by reading and writing to the hardware control registers directly
- How to interact with the RP2040 SDK

The simplest way to learn these tasks is with a Raspberry Pi Pico connected to a Raspberry Pi running the Raspberry Pi OS, a version of Linux. This provides all the tools needed to learn Assembly Language programming. All the software required for this book is open source and readily available on the Raspberry Pi.

This book contains many working programs to play with, use as a starting point, or study. The only way to learn programming is by doing, so don't be afraid to experiment, as it is the only way to learn.

Even if Assembly programming isn't used in your day-to-day life, knowing how the processor works at the Assembly Language level and knowing the low-level binary data structures will make you a better programmer in all other areas. Knowing how the processor works will let you write more efficient C code and can even help with Python programming.

Enjoy your introduction to Assembly Language. Learning it for one processor family helps with learning and using any other processor architectures encountered throughout your career.

Source Code Location

The source code for the example code in the book is located on the Apress GitHub site at the following URL:

`https://github.com/Apress/RP2040-Assembly-Language-Programming`

The code is organized by chapter and includes answers to the programming exercises.

CHAPTER 1

How to Set Up the Development Environment

Microcontrollers like the Raspberry Pi Pico are typically utilized as the brains for smart devices, like microwave ovens, dishwashers, home security systems, weather stations, or irrigation monitors and controllers. At best, they have a small display and perhaps a couple of buttons for taking commands; however, they are still fully functioning computers. The programs that run on them can be quite powerful and sophisticated. Since the microcontroller usually doesn't have a keyboard, mouse, or monitor, we develop their programs on a regular computer, known as a host computer, and then upload the program to the microcontroller to test and finally deploy it.

The Raspberry Pi Pico is a board built around Raspberry's RP2040 ARM CPU chip. Not only is this the heart of the Raspberry Pi Pico, but also Raspberry sells this chip to other manufacturers, including Adafruit, Arduino, Seeed Studio, SparkFun, and Pimoroni. These other companies produce boards like the Raspberry Pi Pico but with different feature sets. For instance, some contain Wi-Fi or Bluetooth functions, easily connect to rechargeable batteries, or are in much smaller form factors. In this book, when we refer to the RP2040, it applies to all the brands of RP2040

S. Smith, *RP2040 Assembly Language Programming*,
https://doi.org/10.1007/978-1-4842-7753-9_1

boards. However, in some cases, we will talk about a specific board, perhaps, because we are discussing Wi-Fi or are referring to specific wiring connections for one board.

Programming the RP2040 in Assembly Language is the main emphasis of this book, but we want to do this by studying real working programs. To do this, we need to hook up our microcontroller to various pieces of hardware. This way we can see programs that perform useful tasks and learn all the flexible and powerful features the RP2040 has to connect to external sensors, controllers, and communication channels. To begin with, we set up the Raspberry Pi Pico on an electronics breadboard, so we can easily wire in the various devices to play with.

This chapter is concerned with physically setting up the Raspberry Pi Pico on a breadboard and wiring it up to a host computer to effortlessly program and debug programs, as well as hook up other components as we encounter them. The *Getting started with Raspberry Pi Pico* guide (from `www.raspberrypi.org/documentation/rp2040/getting-started/`) is an excellent reference on how to do these fundamental tasks. We will not duplicate the contents of the guide; instead, we will point out the important parts that are required for Assembly Language programming, debugging, and playing with the sample programs in this book.

To run most of the programs in this book, you will need

- A Raspberry Pi Pico
- An electronics breadboard
- Pins to attach the Pico to the breadboard
- Miscellaneous connecting wires
- A selection of LEDs
- A soldering iron and solder
- A Raspberry Pi 4 running Raspberry Pi OS

About the Raspberry Pi Pico

The heart of the Raspberry Pi Pico is a new chip developed by Raspberry and ARM. This chip is a system on a chip (SoC) that contains a dual core ARM Cortex-M0+ CPU, 264KB of SRAM, USB port, and support for several hardware devices. Compared to a full computer like the regular Raspberry Pi, the Raspberry Pico lacks a video output port, an operating system, and USB ports for a keyboard and a mouse. But it is possible to connect displays and input devices to the Raspberry Pi Pico, as we'll see later in the book. The specialty connections and input devices aren't used for general-purpose computing; rather, they solve specific problems, such as powering a vending machine and monitoring a greenhouse.

Unlike the CPUs found in desktop and laptop computers, the RP2040 doesn't contain a floating-point unit, vector processing unit, or graphic processing unit. However, one thing it has that regular CPUs lack is a set of eight programmable I/O (PIO) coprocessors. These PIOs have their own Assembly Language and can handle many I/O protocols and tasks independent of the two CPU cores. We'll cover PIOs in Chapter 11. If you already have your RP2040 board wired up and know how to download and debug C programs, then you might want to skip ahead to Chapter 2.

The RP2040 may look underpowered when comparing it to a modern Intel, AMD, or ARM processor, but for the price, it is quite a powerful computer. Table 1-1 compares the RP2040 to some older and newer computers as well as competitors' microcontrollers.

Table 1-1. *Comparison of the Processing Power of the RP2040*

Computer	CPU	Speed (MHz)	Memory (KB)	Bits	Cores
Apple II	MOS 6502	1	48	8	1
IBM PC	Intel 8088	4.77	640	16	1
Arduino Nano	ATmega 328	16	2	8	1
Arduino Due	ARM M3	84	96	32	1
RP2040	**ARM M0+**	**133**	**264**	**32**	**2**
Pi Zero	ARM A53	1024	524,288	32	1
Pi 4	ARM A72	1536	8,388,608	64	4

About the Host Computer

Since microcontrollers don't have a keyboard, a display, or even an operating system, their programs are written on a host computer. For RP2040-based microcontrollers, this could be on a MacOS, Windows, or Linux-based computer. The Raspberry Pi Pico documentation has instructions on how to connect it to all these platforms. The easiest solution is to use a Raspberry Pi 4 as the host vs. using a Windows or Mac computer. Raspberry has made this easy with a complete installation script and clear instructions on how to wire the Raspberry Pi 4 and Raspberry Pi Pico together. The wiring solution of these two boards is the easiest one since the Raspberry Pi 4 already exposes all the necessary pins via its GPIO pins. In this book, we'll use the Raspberry Pi 4, point out the features we will be using, and let you follow the Raspberry-provided documentation to set it up.

How to Solder and Wire

You can't do much with a Raspberry Pi Pico without doing some soldering. Without soldering, you can download programs to the RP2040, flash the onboard LED, and send data back out the USB port to the host computer. However, even to just debug a program, you must do some soldering. The easiest way to set things up is to solder a set of pins to each side of the board, so it can be inserted into an electronics breadboard, which then allows us to connect things up without further soldering. This is great for experimenting. Typically, we would use a new RP2040 board to solder into a final project. At $4 each, there isn't a significant overhead in having a development board and adding new boards to the package when you are finished. To perform debugging requires you to solder pins to the three debugging connections on the end of the board.

The minimum wiring needed is the following three connections between the Pico and the Raspberry Pi 4:

1. Using a micro-USB cable
2. Via the three debugging pins
3. Via a serial port using pins 1, 2, and 3

Don't be scared of soldering; it is actually quite simple and fun. The main trick is to heat up the area where you want the solder to go and touch a bit of solder there. Don't melt it onto the soldering iron's tip and then try to drip it from there. Some vendors provide an option to purchase boards with the pins presoldered for a few dollars extra. Others provide the pins separately, and it is up to you to ensure they are included in your order. Even if the main pins are presoldered, chances are you are going to need to solder pins to the three debug pads. Figure 1-1 shows the wiring, minus the USB cable, of a Raspberry Pi Pico connected to a Raspberry Pi 4.

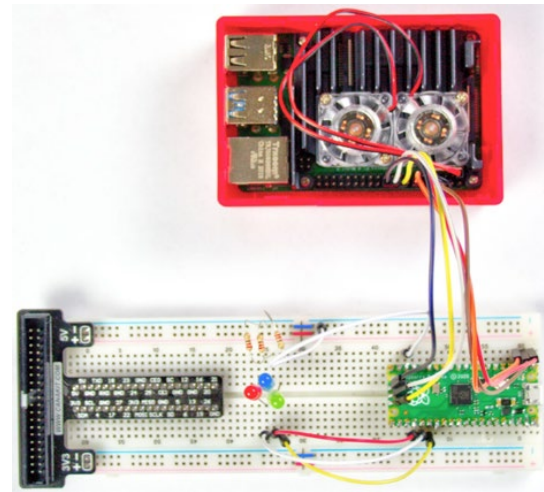

Figure 1-1. *A Raspberry Pi Pico installed in a breadboard and connected to a Raspberry Pi4. The USB cable was removed for clarity. Three LEDs are connected as well.*

Note If you are using an RP2040 board other than the Raspberry Pi Pico, then it is likely that the pins are in different locations on the board, and you will need to adapt the wiring for the location of the pins.

How to Install Software

If you are using a Raspberry Pi as your host computer, then this is straightforward. Use the Raspberry Pi OS as your operating system. This simplifies installation, since it runs 32-bit ARM code and shares development tools with the Raspberry Pi Pico and other RP2040-based boards. The pico_setup.sh script downloads and installs everything required to develop code for RP2040-based systems. As Raspberry's *Getting Started* guide documents, you get pico_setup.sh using wget:

```
wget https://raw.githubusercontent.com/raspberrypi/pico-setup/
master/pico_setup.sh
```

This script sets up both C and Assembly Language programming.

The *Getting Started* guide includes instructions for working with Visual Studio Code, which you are welcome to use, but we won't be covering in this book. This book covers text files that can be edited in any editor, using cmake and make for building, gdb and openocd for debugging, and the minicom for communications.

A Simple Program to Ensure Things Are Working

The easiest way to ensure everything is working is to compile and play with a couple of the SDK examples. The *Getting started with Raspberry Pi Pico* guide walks you through how to do this. Here, rather than duplicate, we'll list the key things you need to be comfortable with, since we will be doing them over and over throughout this book. Here is what you need to know:

1. How to load a program by powering on the Pico while holding down the BootSel button and copying a program to the shared drive
2. How to compile a program to either send its output to the USB or serial port
3. How to use the minicom to display the output that the Pico is sending

4. How to compile a program for debugging
5. How to use openocd and gdb to load and execute a program for debugging

---

Tip Building a program requires running both cmake and make. It isn't always clear which part does what. If you make configuration changes, it is best to delete and recreate the build folder ensuring everything is built from scratch.

Create Some Helper Script Files

When you follow along with the *Getting started with Raspberry Pi Pico* guide, there are quite a few long command lines to type in (or to copy/paste). It saves quite a bit of time to create a collection of small shell scripts to automate the common tasks. You can put these in $HOME/bin and then add

```
export PATH=$PATH:$HOME/bin
```

to the end of the $HOME/.bashrc file. You also need to make these executable with

```
chmod +x filename
```

Next, we need two scripts for minicom—one to listen on the UART and one to listen on the USB, as follows:

File m-uart:

```
minicom -b 115200 -o -D /dev/serial0
```

File m-usb:

```
minicom -b 115200 -o -D /dev/ttyACM0
```

To build debug, I have a script cmaked containing

```
cmake -DCMAKE_BUILD_TYPE=Debug ..
```

To run openocd, ready to accept connections from gdb, I have the script ocdg containing

```
openocd -f interface/raspberrypi-swd.cfg -f target/rp2040.cfg
```

To run gdb-multiarch where the elf file to be debugged is passed as a parameter, I have gdbm containing

```
gdb-multiarch $1
```

When gdb starts, we need to connect to openocd. We can automate this by creating a .gdbinit file in $HOME. This file then contains

```
target remote localhost:3333
```

Note This .gdbinit will be used anytime you start gdb, so if you need to debug a local file without using openocd, then you might want to rename this file while you do that.

Summary

This chapter is the starting point. We haven't done any Assembly Language programming yet, but now we are set up to write, debug, test, and deploy programs written in either C or Assembly Language. The Raspberry Pi Pico is connected to the Raspberry Pi 4 through a USB cable, a serial port, and the debugging port. The Pico is installed in an electronics breadboard ready to have other components connected to it. In Chapter 2, we will use all these tools to start our journey with RP2040 Assembly Language.

CHAPTER 2

Our First Assembly Language Program

Most of the functionality of a Raspberry Pi Pico is contained in the custom RP2040 chip that contains dual core ARM Cortex-M0+ CPUs. The ARM processor was originally developed by a group in Great Britain, who wanted to build a successor to the BBC Microcomputer used for educational purposes. The BBC Microcomputer used the 6502 processor, which was a simple processor with a simple instruction set. The problem was there was no successor to the 6502. They weren't happy with the microprocessors that were around at the time, since they were much more complicated than the 6502 and they didn't want to make another IBM PC clone. They took the bold move to design their own. They developed the Acorn computer that used it and tried to position it as the successor to the BBC Microcomputer. The idea was to use Reduced Instruction Set Computer (RISC) technology as opposed to Complex Instruction Set Computer (CISC) as championed by Intel and Motorola.

Developing silicon chips is an expensive proposition, and unless you can get a good volume going, manufacturing is expensive. The ARM processor probably wouldn't have gone anywhere except that Apple came calling looking for a processor for a new device they had under development—the iPod. The key selling point for Apple was that as the ARM processor was RISC, therefore, it used less silicon than CISC processors and as a result used far less power. This meant it was possible to build a device that ran for a long time on a single battery charge.

S. Smith, *RP2040 Assembly Language Programming*,
https://doi.org/10.1007/978-1-4842-7753-9_2

Unlike Intel, ARM doesn't manufacture chips, it just licenses the designs for others to optimize and manufacture chips. With Apple onboard, suddenly there was a lot of interest in ARM, and several big manufacturers started producing chips. With the advent of smartphones, the ARM chip really took off and now is used in pretty much every phone and tablet and even powers some Chromebooks, making it the number one processor in the computer market.

The designers at ARM are ambitious and architect their processors ranging from low-cost microcontrollers all the way up to the most powerful CPUs used in supercomputers. ARM's line of microcontroller CPUs is the Cortex-M series. We are most interested in the ARM Cortex-M0+ used in Raspberry's RP2040 SoC. To make this chip inexpensive, the transistor count is reduced as much as possible. The M-series CPUs are all 32 bits but have fewer registers and a smaller instruction set than the full A-series ARM CPUs like those used in the full Raspberry Pi. The M-series CPUs are optimized to use as little memory as possible as memory tends to be limited in microcontrollers, again to keep costs down. In this book, we'll see how the Cortex-M0+ works at the lowest level and will often have to deal with the trade-offs made by the chip designers keeping transistor counts down. There are several optional components available from ARM for these chips. We'll consider the ones included in the RP2040, such as the fast integer multiplier and divider (multiplication and division are an extra).

10 Reasons to Use Assembly Language

You can program the Raspberry Pi Pico in MicroPython or C/C++. These are productive languages that hide the details of all the bits and bytes, letting you focus on your application problem. When you program in Assembly Language, you are tightly coupled to a given CPU, and moving your program to another CPU requires a complete rewrite. Each Assembly

Language instruction does only a fraction of the amount of work, so to do anything takes a lot of Assembly Language statements. Therefore, to do the same work as, say, a Python program, takes an order of magnitude larger amount of source code written by the programmer. Writing in Assembly is harder, as you must solve problems with memory addressing and CPU registers that are all handled transparently by high level languages. So why would you ever want to learn Assembly Language programming? Here are ten reasons people learn and use Assembly Language:

1. Even if you don't write Assembly Language code, knowing how the computer works internally allows you to write more efficient code. You can make your data structures easier to access and write code in a style that allows the compiler to generate more efficient code. You can make better use of computer resources like coprocessors and use the given computer to its fullest potential.

2. The PIO coprocessors on the RP2040 are only programmable in Assembly Language. There is a library of common applications in the Software Developer's Kit (SDK), but if you need something beyond these, Assembly Language is the only option.

3. When you are debugging any program on the RP2040 using gdb, a lot of the view you have is at the Assembly Language level. You can see the Assembly Language code generated by the compiler, and you see the CPU registers and can look at raw memory. Understanding this extra level of detail can help you solve the more difficult program bugs. Further, much of the SDK is written in Assembly Language, and you need to know it to step through these parts of the code.

4. To make the RP2040 program faster. If the C compiler or MicroPython runtime isn't producing a program that is responsive enough, then add some Assembly Language code to solve a bottleneck.
5. Interfacing your Pico to a hardware device through the GPIO ports, and the speed of data transfer is extremely sensitive as is how fast the program can process the data. Perhaps, there are a lot of bit level manipulations that are easier to program in Assembly Language.
6. The RP2040 is fast enough to use machine learning. This relies on fast matrix mathematics. If you can make this faster with Assembly Language and/or using the coprocessors, then you can make your AI-based robot or sensor network that much better.
7. Most large programs have components written in different languages. If the program is 99% C++, the other 1% could be Assembly Language, perhaps giving the program a performance boost or some other competitive advantage.
8. Perhaps, you work for a hardware company that makes an RP2040-based board competitor to the Raspberry Pi Pico. These boards have some Assembly Language code in the SDK that must be customized for what you are doing.
9. To look for security vulnerabilities in the Internet of things (IoT) network, you usually need to look at the Assembly Language code; otherwise, you may not know what is really going on and hence where holes might exist.

10. When programming microcontrollers, you have limited memory and resources. Often you need to effectively use every bit to get your application to do what is needed. Often Assembly Language is the only option to cram in every bit of functionality possible.

Computers and Numbers

We typically represent numbers using base 10. The common theory is we do this because we have ten fingers to count with. This means a number like 387 is really a representation for

$$387 = 3 * 10^2 + 8 * 10^1 + 7 * 10^0$$

$$= 3 * 100 + 8 * 10 + 7$$

$$= 300 + 80 + 7$$

There is nothing special about using 10 as our base, and a fun exercise in math class is to do arithmetic using other bases. In fact, the Mayan culture used base 20, perhaps because we have 20 digits—ten fingers and ten toes.

Computers don't have fingers and toes; rather, everything is a switch that is either on or off. As a result, it is natural for computers to use base 2 arithmetic. Thus, to a computer, a number like 1011 is represented by

$$1011 = 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0$$

$$= 1 * 8 + 0 * 4 + 1 * 2 + 1$$

$$= 8 + 0 + 2 + 1$$

$$= 11 \text{ (decimal)}$$

This is great for computers, but we are using four digits for the decimal number 11 rather than two digits. The big disadvantage for humans is that writing out binary numbers is tiring because they take up so many digits.

Computers are incredibly structured, so all their numbers are the same size. When designing computers, it doesn't make sense to have all sorts of differently sized numbers, so a few common sizes have taken hold and become standard.

First of all is the byte, which is 8 binary bits or digits. In our example above with 4 bits, there are 16 possible combinations of 0s and 1s. This means 4 bits can represent the numbers 0 to 15. This means it can be represented by one base 16 digit. Base 16 digits are represented by the numbers 0 to 9 and then the letters A-F for 10-15.

Decimal	0 - 9	10	11	12	13	14	15
Hex Digit	0 – 9	A	B	C	D	E	F

We can then represent a byte (8 bits) as two base 16 digits. We refer to base 16 numbers as hexadecimal. This makes writing out numbers far more compact and easier to deal with.

Since a byte holds 8 bits, it can represent 2^8 (256) numbers. Thus, the byte e6 represents

$$e6 = e * 16^1 + 6 * 16^0$$

$$= 14 * 16 + 6$$

$$= 230 \text{ (decimal)}$$

$$= 1110\ 0110 \text{ (binary)}$$

The ARM Cortex-M0+ processor handles 32-bit numbers; we call a 32-bit quantity a word, and it is represented by 4 bytes. So you might see a string like B6 A4 44 04 as a representation of 32 bits of memory, or one word of memory, or perhaps the contents of one register.

If this is confusing or scary, don't worry. The tools will do all the conversions for you. It's just a matter of understanding what is presented to you on screen. Also, if you need to specify an exact binary number, usually you do so in hexadecimal, though all the tools accept all the formats.

The calculator (galculator) that is bundled with the Raspberry Pi OS, in scientific view, converts between decimal, hex, octal, and binary as well as performs a number of computer-related logical operations. Figure 2-1 shows a a screenshot of this calculator displaying the hex number E6 in binary.

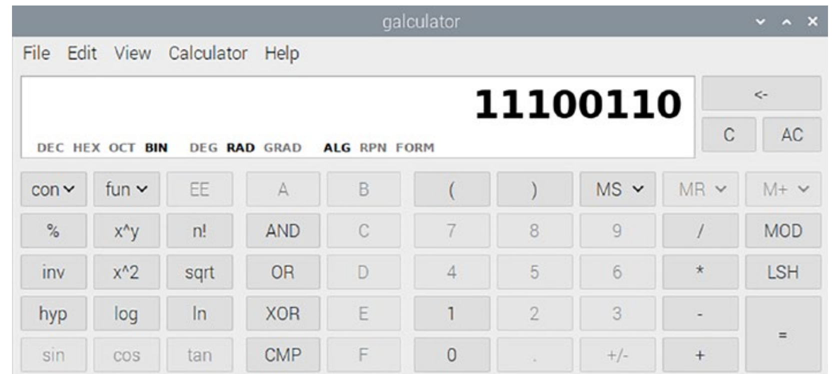

Figure 2-1. *The Raspberry Pi OS's calculator*

There is a bit more complexity in how signed integers are represented and how arithmetic works. We'll cover that a bit later when we go to do some arithmetic.

ARM Assembly Instructions

In this section, we introduce basic architectural elements of the ARM Cortex-M0+ processor and start to look at the form of its machine code instructions. The ARM processor is a Reduced Instruction Set Computer (RISC) that theoretically will make learning Assembly easier. There are fewer instructions, and each instruction is simpler, so the processor can execute each instruction much quicker. The challenge is that it can take quite a few instructions to accomplish fairly easy tasks. As we proceed, we'll provide design patterns to help us combine elements to create larger more sophisticated programs.

If you've programmed an ARM A-series CPU like that in the Raspberry Pi 4 before, then you might know the M-series instruction set as the "thumb" instructions. Newer A-series CPUs typically have 32-bit instructions, but if you want to save memory, there is a "thumb" mode. When you switch to "thumb" mode, most of the instructions are 16 bits in size, thus using half the memory. The M-series CPUs are designed for embedded processors running with minimal memory. This led the designers of the M-series to make the full instruction set to be most of the A-series thumb instructions. In this book, we won't keep referring to them as thumb instructions, since these are the full instruction set of the Cortex-M0+ CPU used in the RP2040. However, you will see references to thumb instructions in the ARM documentation, so it helps to know what they are referring to. Running a simpler instruction set is a key design decision to keep the transistor count, and therefore, the cost and power consumption, of M-series processors down.

In technical computer topics, there are often chicken and egg problems in presenting the material. The purpose of this section is to introduce all the terms and ideas used later. This introduces all the terms, so they are familiar when we cover them in full detail.

CPU Registers

In all computers, data is not manipulated in the computer's memory; instead, it is loaded into a CPU register, and then the data processing or arithmetic operation is performed in the registers. The registers are part of the CPU circuitry allowing instant access, whereas memory is a separate component and there is a transfer time for the CPU to access it.

If you want to add two numbers, you load one into one register and the other into another register, perform the add operation putting the result into a third register, and then copy the answer from the result register into memory. As you can see, it takes quite a few instructions to perform simple operations.

A program on our ARM processor has access to 16 32-bit integer registers and a status register:

- **R0 to R7**: These eight are general purpose that you can use for anything you like.
- **R8 to R11**: These registers can be used to store values, but there are few instructions that can access these directly.
- **R12**: The intraprocedure call scratch register (**IP**).
- **R13**: The stack pointer (**SP**).
- **R14**: The link register. **R14** is used in the context of calling functions, and we'll explain these in more detail when we cover subroutines.
- **R15**: The program counter (**PC**). The memory address of the currently executing instruction.
- **Current Program Status Register (CPSR)**: This special register contains bits of information on the last instruction executed. More on the **CPSR** when we cover branch instructions (if statements).

ARM Instruction Format

Most ARM Cortex-M0+ binary instructions are 16 bits long. There are six 32-bit-long instructions that we'll talk about when we encounter them. Fitting all the information for an instruction into 16 bits is quite an accomplishment requiring using every bit to tell the processor what to do. There are quite a few instruction formats, and we will explain them when we cover that particular instruction. To give you an idea for data processing instructions, let's consider the format for an ADD instruction. The following is the format of the instruction and what the bits specify:

15-9	8-6	5-3	2-0
OpCode	Rm	Rn	Rd

Let's look at each of these fields:

- **Opcode**: Which instruction are we performing, like **ADD** or **SUB**
- **Rm and Rn**: The two registers to add
- **Rd**: The destination register, where to put the result of the addition

For example, consider the Assembly Instruction:

```
ADD R5, R3, R2
```

This is the human-readable form of the instruction to computer **R5** = **R3** + **R2**. The Assembler tool converts this into machine-readable form, namely, the 16 bits: 0x189d. In binary, this is 0001 1000 1001 1101, so if we pull apart the bits, we get

OpCode = 0001100 meaning **ADD**

Rm = 010 = 2 (i.e., **R2**)

Rn = 011 = 3 (i.e., **R3**)

Rd = 101 = 5 (i.e., **R5**)

Note Each register is specified by 3 bits, allowing us to use registers **R0–R7**. If it makes sense to operate on one of the other registers like **SP**, then there will be a specific opcode for that, and you won't specify the register.

If you are used to A-series Assembly Language, this instruction is actually **ADDS**, since it "sets" the **CPSR** when it executes. In M-series Assembly Language, you don't have the option to control whether the **CPSR** is set, so we tend to leave off the S; however, the Assembler will take either.

In A-series Assembly Language, you might see this instruction as **ADD.N** meaning narrow, indicating you want the 16-bit encoding instead of **ADD.W** that gives the 32-bit encoding. Again, the M-series only supports .N, so it isn't necessary to specify this.

When things are running well, each instruction executes in one clock cycle. An instruction in isolation takes three clock cycles, namely, one to load the instruction from memory, one to decode the instruction, and then one to execute the instruction. The ARM CPU is smart and works on three instructions at a time, each at a different step in the process, called the instruction pipeline. If you have a linear block of instructions, they all execute on average taking one clock cycle.

RP2040 Memory

The RP2040 has 264 kilobytes (KB) of memory. Programs are loaded from the Pico's flash storage into memory and executed. The memory holds the program, along with any data or variables associated with it.

- The CPU registers are 32 bits in size. These are used both to address memory and to perform integer arithmetic. This means that memory addresses are 32-bit quantities. This is why we call the RP2040 a 32-bit processor.
- Instructions are mostly 16 bits in size. This doesn't affect the bitness of the processor; it is simply a technique to minimize memory usage and keep CPU processing simple.

If we want to load a register from a known 32-bit memory address, for example, a variable we want to perform arithmetic on, how do we do this? The instruction is only 16 bits in size, and we've already used nearly all the bits to specify the opcode and register to use.

This is a problem that we'll come back to several times, since there are multiple ways to address it. In a CISC computer, this isn't a problem since instructions are typically quite large and variable in length.

You can load from memory by using a register to specify the address to load. This is called indirect memory access. But all we've done is move the problem, since we don't have a way to put the value into that register (in a single instruction).

The quick way to load memory that isn't too far away from the program counter (**PC**) register is to use the load instruction via the **PC**, since it allows an 8-bit offset from the register. This looks like you can efficiently access memory within 256 words of the PC. Yuck, how would you write such code? This is where the GNU Assembler comes in. It lets you specify the location symbolically and will figure out the offset for you.

In Chapter 6, we will look at the details of accessing memory in detail. In all RISC processors, this is a challenge since we need to build 32-bit addresses, but our instructions are only 16 bits in size and can usually only specify 8-bit numbers.

About the GCC Assembler

Writing Assembly Language code in binary as 16-bit instructions would be painfully tedious. Enter GNU's Assembler that gives you the power to specify everything that the ARM can do but takes care of getting all the bits in the right place for you. The general way you specify assembly instructions is

```
label:     opcode    operands
```

The label: is optional and only required if you want the instruction to be the target of a Branch instruction.

There are quite a few opcodes; each one is a short mnemonic that is human readable and easy for the Assembler to process. They include

- **ADD** for Addition
- **LDR** for Load a Register
- **B** for Branch

There are quite a few different formats for the operands, and we will cover those as we cover the instructions that use them.

Hello World

In almost every programming book, the first program is a really simple program to output the string "Hello World." We will do the same with Assembly Language to demonstrate some of the concepts we talked about. We are going to build this sample in the RP2040 SDK framework, which will help us with building the program. The easiest way to do this is to follow their template for projects. First create a "Hello World" folder in your $HOME/pico folder. All the files mentioned here will be placed in this folder. In our favorite text editor, let's create a file "HelloWorld.S" (Listing 2-1).

Listing 2-1. The HelloWorld Program

```
@
@ Assembler program print out "Hello World"
@ using the Pico SDK.
@
@ R0 - first parameter to printf
@ R1 - second parameter to printer
@ R7 - index counter
@

.thumb_func                  @ Necessary because sdk uses BLX
.global main                 @ Provide program starting
                               address to linker

main:
        MOV R7, #0           @ initialize counter to 0
        BL  stdio_init_all   @ initialize uart or usb
loop:
        LDR R0, =helloworld  @ load address of string
        ADD R7, #1           @ Increment counter
        MOV R1, R7           @ Move the counter to second
                               parameter
        BL  printf           @ Call pico_printf
        B   loop             @ loop forever

.data
            .align  4        @ necessary alignment
helloworld: .asciz   "Hello World %d\n"
```


Note It is important that we use .S and not .s in the filename. When we start using more of the SDK, we will need to include some C files. .S will support some C type include files, whereas .s is for pure Assembly Language.

We'll discuss this program in a second, but first we need a file to describe our project to the build system. This file is named CMakeLists.txt; Listing 2-2 shows what it contains..

Listing 2-2. CMakeLists Project Definition File

```
cmake_minimum_required(VERSION 3.13)

include(pico_sdk_import.cmake)
project(HelloWorld C CXX ASM)

set(CMAKE_C_STANDARD 11)
set(CMAKE_CXX_STANDARD 17)

pico_sdk_init()

include_directories(${CMAKE_SOURCE_DIR})

add_executable(HelloWorld
  HelloWorld.S
)

pico_enable_stdio_uart(HelloWorld 1)
pico_enable_stdio_usb(HelloWorld 0)

pico_add_extra_outputs(HelloWorld)

target_link_libraries(HelloWorld pico_stdlib)
```

The CMakeLists.txt file lists our source file, the libraries we need, and some configuration details for the SDK. This file will compile our HelloWorld.S, link it to the pico_stdlib library, and configure the SDK whether to direct the output to either the UART or USB port. There is information on the compiler versions to use, mostly you want to match the SDK requirements since the included parts of the SDK need to be built to be included in our program.

Set one of pico_enable_stdio_uart or pico_enable_stdio_usb to 1 and the other to 0 to control where the output of our "Hello World" text will go.

Copy pico_sdk_import.cmake from the SDK folder pico-sdk/external into our project folder. And finally create a build folder using "mkdir build" or using the file explorer. Your project folder should now look like

```
drwxr-xr-x 6 pi pi 4096 May 23 13:29 build
-rw-r--r-- 1 pi pi  411 May 23 13:29 CMakeLists.txt
-rw-r--r-- 1 pi pi  575 May 23 13:31 HelloWorld.S
-rw-r--r-- 1 pi pi 2763 Apr 10 16:24 pico_sdk_import.cmake
```

Now we are ready to build our project. Open a terminal window and cd into the project folder's build folder. Type

```
cmake ..
```

which will add the SDK files that are needed for this project and create a makefile. Now type

```
make
```

which will compile our project. If all goes well, the build folder should now contain

```
-rw-r--r-- 1 pi pi  18967 May 23 13:29 CMakeCache.txt
drwxr-xr-x 5 pi pi   4096 May 23 13:29 CMakeFiles
-rw-r--r-- 1 pi pi   1570 May 23 13:29 cmake_install.cmake
```



```
drwxr-xr-x 6 pi pi   4096 May 23 13:29 elf2uf2
drwxr-xr-x 3 pi pi   4096 May 23 13:29 generated
-rwxr-xr-x 1 pi pi  22412 May 23 13:29 HelloWorld.bin
-rw-r--r-- 1 pi pi 410911 May 23 13:29 HelloWorld.dis
-rwxr-xr-x 1 pi pi 160532 May 23 13:29 HelloWorld.elf
-rw-r--r-- 1 pi pi 157347 May 23 13:29 HelloWorld.elf.map
-rw-r--r-- 1 pi pi  63101 May 23 13:29 HelloWorld.hex
-rw-r--r-- 1 pi pi  45056 May 23 13:29 HelloWorld.uf2
-rw-r--r-- 1 pi pi  72260 May 23 13:29 Makefile
drwxr-xr-x 6 pi pi   4096 May 23 13:29 pico-sdk
```

HelloWorld.uf2 is our compiled program. We run it by powering off the Raspberry Pi Pico and then powering it on while holding down the BootSel button. In this mode, it will present its flash storage as a shared drive, and we can copy HelloWorld.uf2 onto that drive. As soon as we do this, the Pico will reboot and run our program.

The output can be viewed using minicom; if we created the batch files recommended in Chapter 1, then we run either m-uart or m-usb depending on how the program is configured to run. When we do this, we should observe something like the screenshot shown in Figure 2-2.

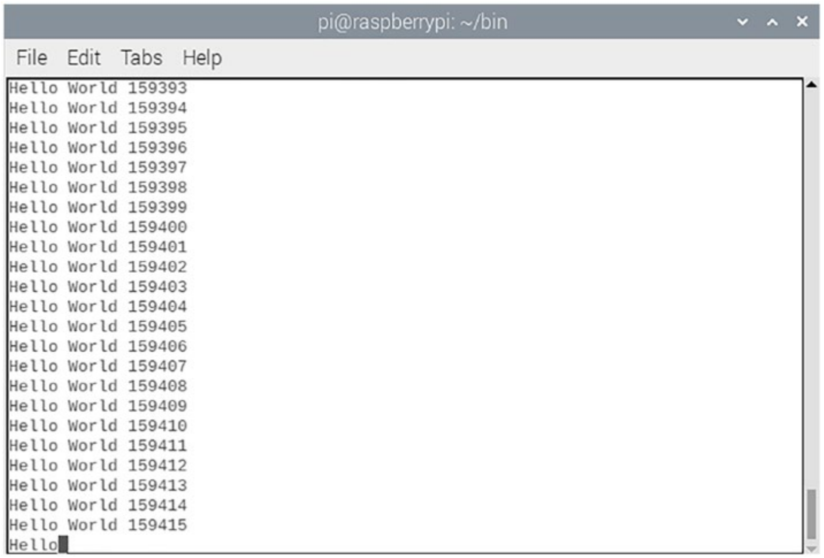

Figure 2-2. *The output from the minicom program for Hello World*

Now that we are running, let's go back and look at the contents of HelloWorld.S.

Our First Assembly Language File

There are four sections to this file, including the header comments, the function definition, the Assembly Language code, and the program data. Let's look at each one of these.

About the Starting Comment

We start the program with a comment that states what it does. We also document the registers used. Keeping track of which registers are doing what becomes important as our programs get bigger.

- Whenever you see a "@" character in a line, then everything after the "@" is a comment. That means it is there for documentation and is discarded by the GNU Assembler when it processes the file.
- Assembly Language is cryptic, so it's important to document what you are doing. Otherwise, you will return to the program after a couple of weeks and have no idea what the program does.
- Each section of the program has a comment stating what it does, and then each line of the program has a comment at the end stating what it does. Everything between a /* and */ is also a comment and will be ignored.

Where to Start

Next, we specify the starting point of our program.

- We need to define this as a global symbol called main that the RP2040 runtime will call to execute our program. All our programs will contain this somewhere.
- We must define this as a thumb_func due to the way the SDK calls our function. We'll look at what this means in Chapter 7. The RP2040 doesn't support any other type of function, but this is still required. If you omit it, you will get a hardware fault when you run the program.
- Our program can consist of multiple .S files, but only one can contain main.

Assembly Instructions

We use five different Assembly Language statements in this example:

1. **MOV**, which moves data into a register. First of all, we use an immediate operand, which start with the '#' sign. So "MOV R7, #0" means move the number 0 into **R7**. In this case, the 0 is in part of the instruction and not stored elsewhere in memory. Secondly, we have "MOV R1, R7," which moves the contents of register **R7** into **R1**. In the source file, the operands can be upper- or lowercase.
2. **BL**, which calls a function. We call two functions: stdio_init_all to initialize communications back to the Raspberry Pi 4 and printf that sends the text. Printf has two parameters in this case: the first is placed in **R0**, which is the address of the string to print, and the second in **R1**, which is the integer counter.
3. **LDR**, which is used to both load memory addresses and load the contents for memory. In this case, we use "LDR R0, =helloworld" that loads register 0 with the address of the string we want to print.
4. **ADD**, which adds two 32-bit integers. "ADD R7, #1" adds the immediate operand #1 (the number 1) to register **R7** incrementing it.
5. **B**, which branches to the label loop. Labels are symbolic indicators of positions in the code or data.

Next up is the last section, the data section.

Data

Next, we have .data that indicates the following instructions are located in the data section of the program:

- First, we have an ".align 4" statement. This ensures the memory address is divisible by four. Some instructions require the data to be aligned, and even if the instruction doesn't require data alignment, data loads faster when it is aligned (the memory circuitry usually will require two reads for a nonaligned 32-bit quantity).
- In this, we have a label "helloworld" followed by an .asciz statement and then the string we want to print.
- The .asciz statement tells the Assembler just to put our string in the data section, and then we can access it via the label as we do in the **LDR** statement. The z in asciz asks the Assembler to place a 0 byte after the last character, which is required by the printf function. We'll talk later about how text is represented as numbers, the encoding scheme here being called ASCII.
- The last "\n" character is how we represent a new line.

These are the individual instructions; now we'll discuss how they work together.

Program Logic

On full computers running operating systems like Linux, Windows, or MacOS, programs usually run, do their job, and then terminate returning control to the operating system. In this way, many programs are run all under the control of the operating system, and the operating system is the only program that runs from power on to power off. On microcontrollers, typically, there is no operating system. The only thing that runs is our program. The expectation is our program will be run shortly after the RP2040 powers on and then terminated when it is powered off. This is why we have created an infinite loop that runs forever, which is typical of most microcontroller programs.

If we terminated the program after printing "Hello World," the CPU would halt until the RP2040 is powered off and on again. Chances are we would miss the printing of "Hello World" because we didn't start minicom fast enough. I added the counter as a simple example and so that when you run minicom, you can see something actually happening, namely, the count forever increasing till it wraps around and starts over.

The call stdio_init_all at the beginning initializes either the UART or USB channel depending on what we configured in our CMakeLists.txt file. The call to printf is an alias to pico_printf which is an implementation of the C runtime's printf but contained in the RP2040 SDK for anyone to use. As Assembly Language programmers, we can call pretty much anything as long as we know the protocol to do so.

You might wonder why we keep our count in register **R7** rather than using **R1** and saving having to move **R7** into **R1** before each call to printf. The reason is that there is a register usage protocol when calling functions and **R1** is allowed to be used by printf, without printf saving whatever we put there. If printf uses **R7**, then it has to save our value and restore it before returning. We will study the register usage protocol in Chapter 7.

The printf function takes a variable number of arguments; the first argument is always a string. If the string contains certain characters like %d, this means print a number, which then causes printf to look for a second parameter containing a 32-bit integer. This is handy for us, since it converts the binary 32-bit quantity into human-readable numbers for us. Hopefully, you are familiar with C programming, and this is all basic.

Reverse Engineering Our Program

We talked about how each Assembly Language instruction is compiled into a 16-bit number. The Assembler did this for us, but can we see what it did? To do so, we look at the HelloWorld.dis file that was generated in our build folder. This file contains everything that is combined to create our program. This includes the code to initialize the RP2040 from the SDK, the code for the printf function, as well as the code to communicate with either the UART or USB ports. Listing 2-3 contains only our code and data sections.

Listing 2-3. Disassembly of Hello World

```
1000035c <main>:
1000035c:       2700        movs r7, #0
1000035e:       f004 fecd   bl   100050fc <stdio_init_all>

10000362 <loop>:
10000362:       4803        ldr  r0, [pc, #12]; (10000370
                                 <loop+0xe>)
10000364:       3701        adds r7, #1
10000366:       1c39        adds r1, r7, #0
10000368:       f004 febc   bl   100050e4 <__wrap_printf>
1000036c:       e7f9        b.n  10000362 <loop>
1000036e:       0000        .short 0x0000
10000370:       20000180    .word 0x20000180
...
20000180 <helloworld>:
20000180:       6c6c6548    .word 0x6c6c6548
20000184:       6f57206f    .word 0x6f57206f
20000188:       20646c72    .word 0x20646c72
2000018c:       000a6425    .word 0x000a6425
```

In Listing 2-3, the first column is the memory address where the item will be located. The second column is the binary form of the instruction created by the Assembler from the human-readable forms of the instruction and its operands that are in the next two columns. The disassembler sometimes adds helpful comments in angle brackets <> or after a semicolon.

Some points to notice from this listing:

- Most of the instructions compile to 16-bit quantities except for the **BL** statements that are 32 bits. Practically speaking, if the M0+ CPU insisted on making **BL** statements 16 bits, then you would need to build the address in a register and then jump to it indirectly, which would take several statements. This way we can efficiently call functions with only one Assembly Language statement.
- **MOV** and **ADD** have been changed to **MOVS** and **ADDS**; this is to indicate that these set the **CPSR**. The GNU toolchain is used for both ARM M-series and A-series processors, and we see features of the A-series processor being represented, even though we can't change this option on the M-series CPU.
- The branch statement **B** has been changed to **B.N**. This is to indicate this is the 16-bit version of this instruction. There is a 32-bit version of this instruction, **B.W**, and the Assembler will use **B.W** if the target of the branch is too far away to fit in 16 bits. Hence, we don't need to worry about this; the Assembler will use the most efficient version it can.

- Notice the second **MOV** statement was changed to "adds r1, r7, #0". This adds **R7** to 0 and puts the result in **R1**, which is what we want. With only 16 bits, we can't waste any bits with duplicate functions, so if there are ever two ways to do something, one is aliased to the other. Again, the Assembler does these substitutions for us, so we don't need to remember all these tricks that go on under the covers.

Look at the **LDR** instruction; it changed from

```
ldr R0, =helloworld
```

to

```
ldr r0, [pc, #12]; (10000370 <loop+0xe>)
```

This is the Assembler helping you with the ARM processor's obscure mechanism of addressing memory. It lets you specify a symbolic address, namely, "helloworld," and translate that into an offset from the program counter. I'm certainly happy to have a tool to do that bit of nastiness for me.

Note [pc, #12] points to a bit of memory that holds 20000180, which is the actual address of our "Hello World" string. The Assembler inserted this for us, and we'll cover this in detail in Chapter 6.

If you count the bytes, our Assembly Language program has 18 bytes of code and 22 bytes of data, which is pretty small. This is the power of the small 16-bit Assembly instructions used in the ARM Cortex-M0+. Notice that the uf2 file is 45k long and the size of the code it contains is about 22k. This is because in addition to our code, it contains the SDK runtime code to initialize the RP2040, set up the environment, and then run our program. It also contains the SDK code for printf and any other SDK routines that are used. This is the total code running in the 264KB of memory available to the RP2040. There is nothing else, no operating system. Everything running is compiled from source code into the UF2 file, and that is all that is running on the RP2040 after it powers up. A bit of code in the RP2040 firmware loads our code into memory and then passes execution to it and then away we go.

Summary

In this chapter, we introduced the ARM Cortex-M0+ processor and Assembly Language programming along with why we want to use Assembly Language. We covered the tools we will be using. We covered how computers represent positive integers. We then looked in more detail at how the ARM CPU represents Assembly Language instructions along with the registers it contains for processing data. We introduced the RP2040's memory. We introduced the GNU Assembler that will assist us in writing our Assembly Language programs. We then created a simple complete program to print "Hello World!" and viewed it in minicom on the Raspberry Pi. In Chapter 3, we will look in more detail at the tools used to build and debug programs.

Exercises

2-1. Convert the decimal number 1234 to both binary and hexadecimal.

2-2. Download the source code for this book from GitHub and compile the HelloWorld program on your Raspberry Pi. Next, run it on your RP2040 board and observe the output in minicom.

2-3. Compare the size of the uf2 file when you set the various output options between none, UART, and USB. Remember to delete the build folder whenever you change the CMakeLists.txt file. Which one is the better option as your program size approaches 264KB?

2-4. Decode a couple of the binary format of the instructions in Listing 2-3 to see if you can figure out the operand and where the registers are specified.

2-5. Change the string that is printed. Can you print the number in hexadecimal?

2-6. Rather than count up, change the program to count down subtracting 1 rather than adding 1 in the loop.

CHAPTER 3

How to Build and Debug Programs

In this chapter, we look in more detail at the build tools we are using. The RP2040 SDK does much of the work supporting building our programs, but it is beneficial to understand what is going on underneath the high-level tools. Next, we delve into the GNU debugger (**gdb**), which single-steps through our programs and examines registers and memory as we go.

CMake

CMake is an open source, build automation tool that is cross-platform and compiler independent. The goal of using CMake in the RP2040 SDK is to hide the messy details of using the various compiler toolchains on the host computer, whether it's a Raspberry Pi, Windows, or MacOS. With CMake, your project is built from the same CMakeLists.txt file, and you don't need to know the details of how to run the GNU Assembler. Within the RP2040 SDK, there is experimental support for the LLVM CLang toolchain, as well as the official support for the GNU toolchain. We'll only address the GNU tools in this book, but CMake hides most of the differences if you want to experiment. To fully cover CMake requires a full book in itself, so we are only covering what we need to know for our Assembly Language programming.

S. Smith, *RP2040 Assembly Language Programming*,
https://doi.org/10.1007/978-1-4842-7753-9_3

CMake knows about the main C compilers and Assemblers, including building C and Assembly Language files using the GNU toolchain. The SDK adds CMake files to give specific options, like we are compiling for the ARM Cortex-M0+ processor, and lets CMake know where all the SDK files are located. We'll go into this in more detail later in this chapter. The goal is to specify our target executable name and list our files that need to be built; then CMake, with the help of some definition files in the SDK, does all the work. CMake doesn't actually build our product; instead, it creates a makefile for the GNU Make tool which we'll cover in the next section. GNU Make is then run to do the compiling.

Make doesn't know anything about compiler tools; instead, it has a list of rules that specify commands to run which CMake created. Now we'll go through a CMakeLists.txt file based on the one in Listing 2-2, but with a couple of instructions added to statements to highlight features we haven't talked about yet.

```
cmake_minimum_required(VERSION 3.13)
```

The preceding line specifies the minimum version of CMake required to build the project. This is the recommended value from the SDK and indicates the minimum version to build the SDK files.

```
include(pico_sdk_import.cmake)
```

The include statement includes the code from the specified file into our file and executes it. When we set up our folder, we copied the pico_sdk_import.cmake file into the same place as our CMakeLists.txt file. Pico_sdk_import.cmake checks that the environment variable PICO_SDK_PATH is set and then includes ${PICO_SDK_PATH}/pico_sdk_init.cmake. This file then includes several further files that set up all the rules for building the SDK files and applies all the configurable options documented in the SDK's reference manual.

```
project(HelloWorld C CXX ASM)
```

The preceding line defines our project name as HelloWorld and that we will use C, C++, and Assembler. Even though we didn't use C or C++, many such files are included in the SDK.

```
set(CMAKE_C_STANDARD 11)
set(CMAKE_CXX_STANDARD 17)
```

The preceding statements define the version of the language used (not the version of the compiler). For instance, we are using C11 (or more formally ISO/IEC 9899:2011). These are the minimum versions of the languages required for the SDK to work.

```
pico_sdk_init()
```

The preceding call executes a macro to set up the SDK.

```
include_directories(${CMAKE_SOURCE_DIR})
```

The preceding call sets up where to look for include directories. If unchanged, this call includes all the various source files in the SDK. If your project has the source code spread over multiple folders, then you can add them here separated by spaces.

```
add_executable(HelloWorld
  HelloWorld.S
  cfile.c
  cplusplusfile.cpp
)
```

The preceding statement is where to add source files.

Note They can be of different types, for example, a C and a C++ file. Based on the file extension, cmake creates the correct build rules into the generated makefile. Usually, as the project grows, all we need to do is add files here, and cmake will take care of the rest.


```
pico_enable_stdio_uart(HelloWorld 1)
pico_enable_stdio_usb(HelloWorld 0)
```

The preceding macros are defined in the Pico's SDK. We set them to control where the output from printf statements goes. Set the second parameter to 1 to enable the device and 0 to disable it.

Note Change the options here and rebuild, rather than modifying the source code. The correct code to support either the UART or USB port is included when our project is built.

```
pico_add_extra_outputs(HelloWorld)
```

If we leave the preceding line out, the build works, and an .elf file is produced, which is an executable file for Linux; however, this isn't always what we want. The pico_add_extra_outputs statement causes cmake to generate build rules to create a .uf2 file from the .elf file, which is the correct file to copy to the Raspberry Pi Pico's flash storage. It also generates useful files like the .dis file (disassembly file).

```
target_link_libraries(HelloWorld pico_stdlib)
```

The preceding statement specifies the libraries to use. The library we've needed so far is pico_stdlib, but we can add other libraries as we need them.

GNU Make

GNU Make is a tool used to build programs by taking a number of rules for how to compile programs and executing them. The rules are in the form of dependencies and make compares the dates of the files, so if the dependent file is newer than what it depends upon, then it knows to not do that step.

Working with make is more efficient than working with shell scripts, since it only builds what changed, therefore building programs more quickly. We won't be writing dependency scripts or makefiles ourselves; instead, cmake will write them for us. However, sometimes, we need to know what is happening at a lower level, namely, what command line arguments are passed to the Assembler, and this is a good place to look.

GNU Make has the following functions:

1. It specifies the rules for how to build one thing from another.
2. It examines the file date/times to determine what needs to be built.
3. It issues commands to build the components.

For instance, in the makefile created by cmake for our "Hello World" program, we see many calls to cmake and make on all sorts of things such as updating the progress meter and compiling various SDK files. If we want to see how our HelloWorld.S file is assembled, HelloWorld.S.obj is built by running make on the file CMakeFiles/HelloWorld.dir/build.make

```
HelloWorld.S.obj:
      $(MAKE) -f CMakeFiles/HelloWorld.dir/build.make
CMakeFiles/HelloWorld.dir/HelloWorld.S.obj
```

If we delve into build.make, we find

```
CMakeFiles/HelloWorld.dir/HelloWorld.S.obj: ../HelloWorld.S
      @$(CMAKE_COMMAND) -E cmake_echo_color --switch=$(COLOR)
      --green --progress-dir="/home/pi/pico/Chapter 2 Hello
      World/build/CMakeFiles" --progress-num=$(CMAKE_
      PROGRESS_1) "Building ASM object CMakeFiles/HelloWorld.
      dir/HelloWorld.S.obj"
```



```
      /usr/bin/arm-none-eabi-gcc $(ASM_DEFINES) $(ASM_
      INCLUDES) $(ASM_FLAGS) -o CMakeFiles/HelloWorld.dir/
      HelloWorld.S.obj   -c "/home/pi/pico/Chapter 2 Hello
      World/HelloWorld.S"
```

This says CMakeFiles/HelloWorld.dir/HelloWorld.S.obj depends on HelloWorld.S, meaning if HelloWorld.S is newer than HelloWorld.S.obj, then we execute these build rules. The first rule calls cmake, and this is to print a green status line showing our progress in the build but ignore this. The second line is the command line for the Assembler arm-none-eabi-gcc with its command line arguments.

Make extensively uses variables, both internal and environmental. The main command line flags are contained in $(ASM_FLAGS). This is useful since to change the command line arguments, we can search for this and see where cmake sets it and which cmake variable to change to affect it.

To see all these variables expanded, update HelloWorld.S so it needs compiling and then run

```
make -n
```

This won't execute the build but prints out all the commands that would be executed, and we can see the exact call to the GNU Assembler (with the include folders removed for brevity):

```
/usr/bin/arm-none-eabi-gcc -DCFG_TUSB_DEBUG=0 -DCFG_TUSB_MCU=
OPT_MCU_RP2040 -DCFG_TUSB_OS=OPT_OS_PICO -DPICO_BIT_OPS_PICO=1
-DPICO_BOARD=\"pico\" -DPICO_BUILD=1 -DPICO_CMAKE_BUILD_
TYPE=\"Release\" -DPICO_COPY_TO_RAM=0 -DPICO_CXX_ENABLE_
EXCEPTIONS=0 -DPICO_DIVIDER_HARDWARE=1 -DPICO_DOUBLE_PICO=1
-DPICO_FLOAT_PICO=1 -DPICO_INT64_OPS_PICO=1 -DPICO_MEM_OPS_
PICO=1 -DPICO_NO_FLASH=0 -DPICO_NO_HARDWARE=0 -DPICO_ON_
DEVICE=1 -DPICO_PRINTF_PICO=1 -DPICO_STDIO_USB=1 -DPICO_TARGET_
```

```
NAME=\"HelloWorld\" -DPICO_USE_BLOCKED_RAM=0  -mcpu=cortex-
m0plus -mthumb -O3 -DNDEBUG   -ffunction-sections
-fdata-sections -o CMakeFiles/HelloWorld.dir/HelloWorld.S.obj
-c "/home/pi/pico/Chapter 2 Hello World/HelloWorld.S"
```

As Assembly Language programmers, we like complete control over what we are doing and don't like tools doing work for us behind our backs. We can't edit these makefiles as they are generated by cmake, and anything we do will be overwritten; furthermore, we often delete and recreate the build folder for cmake changes to take effect. The awareness of make for RP2040 development is to double-check that cmake is doing what we think it is doing when we are trying to solve build issues.

Now that we know more about the build process, we will advance to techniques for debugging our programs.

Print Statements

We can perform many debugging type functions peppering our source code with calls to the SDK's printf function. The SDK's printf is quite lightweight compared to the full C runtime printf function, because it doesn't use memory allocation and is reentrant; even so, it contains most of the functionality that C programmers typically use. In our "Hello World" program, adding printf was easy and nondisruptive since we used only one register. However, there are a few complexities to be aware of:

- Functions are allowed to use registers **R0–R3** without saving them. If we use any of these four registers, then save them before calling printf and restore them afterward. Furthermore, printf disrupts the **CPSR**, meaning it can't be inserted in the middle of something relying on the **CPSR**.

- Each time we want to see something new, we need to add a printf call. Add code to set registers and call the function. Then we need to recompile, copy the .uf2 file to the Pico, and observe the output.
- There is only 264KB of memory on the RP2040, and creating a lot of strings to print things can use a substantial amount of this memory.
- Even though the SDK is lightweight, it still takes memory and adds processing time to our program, perhaps disrupting timing-sensitive tasks.
- Adding and removing source code for the printf statements could result in bugs, for example, if we make a mistake and delete one instruction too many.
- There may be surprising side effects from executing printf that disrupt your program.

Some of these problems can be alleviated by using the GNU Assembler's macro feature. We'll look at how to do this in Chapter 7; in addition, printf is a useful function, but to address these limitations, what we really need is a full debugger and this is the GNU debugger (**gdb**).

GDB

When programming with Assembly Language, being proficient with the debugger is critical to success. Not only will this help with your Assembly Language programming, but also it is a great tool for you to use with your high-level language programming. **Gdb** addresses many of the concerns with printf mentioned above; however, it introduces a few of its own and is a technical tool that requires a learning curve to become proficient with it.

Gdb was installed by the pico_setup.sh script. To use **gdb**, wire up the debug ports on your RP2040 board as indicated in Chapter 1. Also, have the UART pins wired up and use that for print statements. This is because when you stop program execution with **gdb**, it stops the processor, and this disconnects the USB port.

Preparing to Debug

The GNU debugger (**GDB**) can debug your program as it is, but this isn't the most convenient way to go. In our HelloWorld program, we have the label "helloworld." If we debug the program as is, the debugger won't know anything about this label, since the Assembler changed it into an address in a .data section. There is a command line option for the Assembler that includes a table of all our source code labels and symbols, so we can use them in the debugger. This makes our program executable a bit larger. We don't need to know the Assembler command line argument; instead, we tell cmake we want a debug build.

Often, we set debug mode while we are developing the program and then turn off debug mode before releasing the program. Unlike some high-level programming languages, debug mode doesn't affect the machine code that is generated, so the program behaves exactly the same in both debug and nondebug mode.

We don't want to leave the debug information in our program for release, because besides making the program executable larger, it is a wealth of information for hackers to help them reverse engineer the program. If you are creating an open source program, then this isn't important as anyone can look at the source code and build the program with any options they like. There are several cases where hackers caused mischief because the program still had debugging information present.

Note Make sure the CMakeLists.txt is configured to output to the UART and not the USB port. When **gdb** halts the CPU, the USB connection is broken.

To add debug information to our program, we invoke cmake setting the CMAKE_BUILD_TYPE to Debug. To ensure everything is generated properly, we delete and recreate the build folder first:

```
rm -rf build
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Debug ..
make
```

Note We could have used the cmaked script from Chapter 1, so we don't need to remember the cmake command line argument for a debug build.

Now we are all set to continue development in debug mode.

Beginning GDB

Before starting the debugger, we need to run

```
openocd -f interface/raspberrypi-swd.cfg -f target/rp2040.cfg
```

or use the ocdg script created in Chapter 1.

To start debugging our "Hello World" program, enter the command

```
gdb-multiarch HelloWorld.elf
```

Or use our script from Chapter 1:

```
gdbm HelloWorld.elf
```

This yields the abbreviated output:

```
GNU gdb (Raspbian 8.2.1-2) 8.2.1
Copyright (C) 2018 Free Software Foundation, Inc.
...
warning: No executable has been specified and target does not
support
determining executable automatically.  Try using the "file"
command.
0x10005128 in ?? ()
Reading symbols from HelloWorld.elf...done.
(gdb)
```

The warning is a side effect that we are programming a microcontroller and there is no operating system. It means we aren't ready to run our program yet; we need to enter one more command to load it first.

Note If we didn't create a .gdbinit file as indicated in Chapter 1, then we need to enter the command "target remote localhost:3333" at this point to connect to the RP2040 board.

- gdb is a command line program.
- (gdb) is the command prompt where you type commands.
- (hit tab) for command completion. Enter the first letter or two of a command as a shortcut.

First, we have to load the program; type

```
load
```

(or lo for short). We can do this repeatedly, so in another terminal window, we can make changes to the program, recompile it, and load it again. This way we don't need to restart the **gdb** environment and redo any commands we've done. Raspberry recommends issuing a "monitor reset init" command after load, which is a good idea, even if it isn't always necessary.

To make the program run, type

```
continue
```

(or c for short).

As long as you run minicom configured to read the uart (m-uart), you will see the "Hello World" strings going by. The program will run forever, but you can stop its execution by typing control-c.

After terminating the program, we will either be inside our code or inside one of the RP2040 SDK's routines.

To start in our routine, set a breakpoint and stop in our main routine. Do this by using the breakpoint command (or b):

```
b main
```

Now reset and rerun with

```
monitor reset init
continue
```

The result is

```
Continuing.
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x0000012a msp: 0x20041f00
```

```
Thread 1 hit Breakpoint 1, main ()
    at /home/pi/pico/Chapter 2 Hello World/HelloWorld.S:14
14 MOV R7, #0 @ initialize counter to 0
```

As far as a **gdb** is concerned, the whole .elf file is our program, including the SDK code to initialize the RP2040. Since the entire SDK is provided as source code, anything that is described here for debugging our code works equally well for the SDK code. The provision is that you need to let the SDK code do initial setup on the RP2040 before a breakpoint can actually stop the CPU.

To list our program, type

```
list
```

(or l).

This lists ten lines. Type

```
l
```

for the next ten lines. Type

```
list 1,1000
```

to list our entire program.

The list gives us the source code for our program, including comments. This is a handy way to find line numbers for other commands. If we want to see the raw machine code, we can have **gdb** disassemble our program with

```
disassemble main
```

This shows the actual code produced by the Assembler with no comments.

We can step through the program with the step command (or s). As we go, we want to see the values of the registers. We get these with info registers (or i r):

```
Thread 1 hit Breakpoint 1, main ()
    at /home/pi/pico/Chapter 2 Hello World/HelloWorld.S:14
14 MOV R7, #0 @ initialize counter to 0
(gdb) s
15 BL stdio_init_all @ initialize uart or usb
(gdb) i r
r0             0x200002b0          536871600
r1             0x1000035d          268436317
r2             0x200001e0          536871392
r3             0x200002b0          536871600
r4             0x10000264          268436068
r5             0x20041f01          537140993
r6             0x18000000          402653184
r7             0x0                 0
r8             0xffffffff          -1
r9             0xffffffff          -1
r10            0xffffffff          -1
r11            0xffffffff          -1
r12            0x34000040          872415296
sp             0x20042000          0x20042000
lr             0x10000223          268436003
pc             0x1000035e          0x1000035e <main+2>
xPSR           0x61000000          1627389952
msp            0x20042000          0x20042000
psp            0xfffffffc          0xfffffffc
primask        0x0                 0
basepri        0x0                 0
faultmask      0x0                 0
control        0x0                 0
```

We see **R7** was set to 0 as expected. We can continue stepping or enter continue (or c) to continue to the next breakpoint if there is one. We can set as many breakpoints as we like. We can see them all with the info breakpoints (or i b) command. Delete a breakpoint with the delete command, specifying the breakpoint number to delete.

```
(gdb) i b
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x1000035c /home/pi/pico/Chapter
                                           2 Hello World/
                                           HelloWorld.S:14
breakpoint already hit 4 times
(gdb) delete 1
(gdb) i b
No breakpoints or watchpoints.
(gdb)
```

We haven't dealt with memory much, but **gdb** has good mechanisms to display memory in different formats. The main command being x with the format

```
x /Nfu addr
```

where

- N is the number of objects to display
- f is the display format where some common ones are
  - t for binary
  - x for hexadecimal
  - d for decimal
  - i for instruction
  - s for string

- u is unit size and is any of
  - b for bytes
  - h for halfwords (16 bits)
  - w for words (32 bits)
  - g for giant words (64 bits)

The main routine is stored at memory location 0x1000035c:

```
(gdb) x /4ubft main
0x1000035c <main>: 00000000 00100111 00000011 11110000
(gdb) x /4ubfi main
   0x1000035c <main>: movs r7, #0
=> 0x1000035e <main+2>: bl 0x10003d9c <stdio_init_all>
   0x10000362 <loop>: ldr r0, [pc, #12] ; (0x10000370 <loop+14>)
   0x10000364 <loop+2>: adds r7, #1
(gdb) x /4ubfx main
0x1000035c <main>: 0x00 0x27 0x03 0xf0
(gdb) x /4ubfd main
0x1000035c <main>: 0 39 3 -16
```

To exit **gdb**, type q (for quit or type control-d).

Table 3-1 provides a quick reference to the **GDB** commands introduced in this chapter. As we learn new things, we'll add to our knowledge of **gdb**. It is a powerful tool to help us develop our programs. Assembly Language programs are complex and subtle, and **gdb** is great at showing us what is going on with all the bits and bytes.

Table 3-1. *Summary of Useful GDB Commands*

Command (short form)	Description
break (b) line	Set breakpoint at line
continue (c)	Continue running the program
step (s)	Single-step program
quit (q or control-d)	Exit gdb
info registers (i r)	Print out the registers
control-c	Interrupt the running program
info break (i b)	Print out the breakpoints
delete n	Delete breakpoint n
x/Nuf expression	Show contents of memory
load (lo)	Load the program
monitor reset init (mon reset init)	Reset GDB

It's worthwhile to single-step through the "Hello World" sample program and examine the registers at each step to ensure you understand what each instruction is doing.

Even if you don't know of a bug, many programmers like to single-step through their code to look for problems and to convince themselves that their code is correct. Often, two programmers do this together as part of the pair programming agile methodology.

Summary

In this chapter, we introduced the GNU Make program that we will use to build our programs. This is a powerful tool used to handle all the rules for the various compilers and linkers we need.

We then introduced the GNU debugger, which will allow us to troubleshoot our programs. Unfortunately, programs have bugs, and we need a way to single-step through them and examine all the registers and memory as we do so. **GDB** is a technical tool, but it's indispensable in figuring out what our programs are doing.

In Chapter 4, we will look at loading data into the CPU registers and performing basic arithmetic. We'll see how negative numbers are represented and learn new techniques for manipulating binary bits.

Exercises

3-1. Step through the "Hello World" program from Chapter 2 to ensure you understand the changes each instruction makes to the registers. Ensure you can see the output of the print statements.

3-2. Experiment with the various **gdb** commands to ensure you are familiar with their various options.

3-3. Why does CMake generate a makefile that you use to build your program rather than building it itself?

CHAPTER 4

How to Load and Add

In this chapter, we will go slowly through the **MOV**, **ADD**, and **SUB** instructions to lay the groundwork on how they work, especially in the way they handle parameters (operands). In the following chapters, we can proceed at a faster pace as we encounter the rest of the ARM instruction set. Before getting into the **MOV**, **ADD**, and **SUB** instructions, we will discuss the representation of negative numbers and the concepts of shifting and rotating bits.

About Negative Numbers

In the previous chapter, we discussed how computers represent positive integers as binary numbers, called unsigned integers, but what about negative numbers? Our first thought might be to make one bit represent whether the number is positive or negative. This is simple but requires extra logic to implement, since now the CPU must look at the sign bits and then decide whether to add or subtract and in which order.

About Two's Complement

The great mathematician John von Neumann of the infamous Manhattan Project came up with the idea of the **two's complement** representation for negative numbers, in 1945, when working on the Electronic Discrete Variable Automatic Computer (EDVAC)—one of the earliest electronic computers.

S. Smith, *RP2040 Assembly Language Programming*,
https://doi.org/10.1007/978-1-4842-7753-9_4

Consider a 1-byte hexadecimal number like 01. If we add

```
0x01 + 0xFF = 0x100
```

(all binary ones), we get 0x100.

However, if we are limited to 1-byte numbers, then the 1 is lost, and we are left with 00:

```
0x01 + 0xFF = 0x00
```

The mathematical definition of a number's negative is a number that when added to it makes zero; therefore, mathematically, FF is -1. You can get the two's complement form for any number by taking

2^N - number

In our example, the two's complement of 1 is

2^8 - 1 = 256 - 1 = 255 = 0xFF

This is why it's called two's complement. An easier way to calculate the two's complement is to change all the 1s to 0s and all the 0s to 1s and then add 1. If we do that to 1, we get

```
0xFE + 1 = 0xFF
```

Two's complement is an interesting mathematical oddity for integers that are limited to having a maximum value of one less than a power of two, which is all computer representations of integers.

Why would we want to represent negative integers this way on computers? As it turns out, addition is simple for the computer to execute. There are no special cases; if you discard the overflow, everything works out. This means less circuitry is required to perform the addition, and as a result, it can perform faster. Besides handling the signs correctly, this also results in the CPU using the same addition logic for signed and unsigned arithmetic—another circuitry-saving measure. Consider

```
5 + -3
```

3 in 1 byte is 0x03 or 0000 0011 binary.

Inverting the bits is

```
1111 1100
```

Add 1 to get

```
1111 1101 = 0xFD
```

Now add

```
5 + 0xFD = 0x102 = 2
```

Since we are limited to 1 byte or 8 bits, we truncate the leading 1 and are left with 2.

About Raspberry Pi OS Calculator

Fortunately, we have computers to do the conversions and arithmetic for us, but when we see signed numbers in memory, we need to recognize what they are. The Raspberry Pi OS calculator calculates two's complement for you. Type the negative number in decimal and then press the HEX button. Figure 4-1 shows the Raspberry Pi OS calculator representing -3 as a 32-bit hexadecimal number.

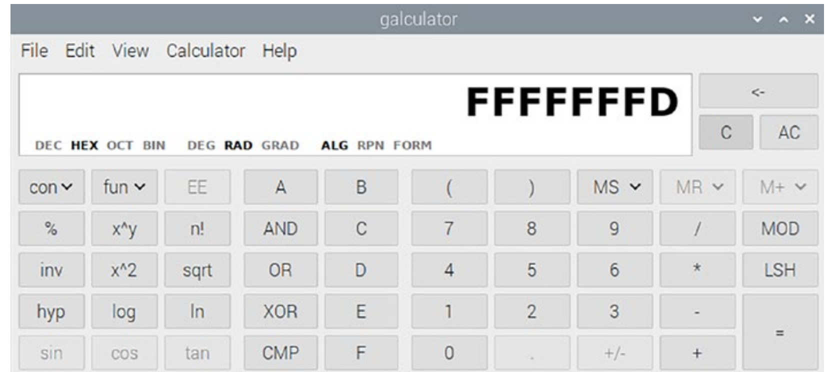

Figure 4-1. *The Raspberry Pi OS calculator shows the two's complement of 3*

About One's Complement

If we don't add 1 and just change all the 1s to 0s and vice versa, then this is called **one's complement**. There are uses for the **one's complement** form, and we will encounter this again in later chapters.

Big- vs. Little-Endian

If we look at a 32-bit representation of 1 stored in memory, it is

```
01 00 00 00
```

rather than

```
00 00 00 01
```

Most processors pick one format or the other to store numbers. Motorola and IBM mainframes use what is called Big-Endian, where numbers are stored in the order of most significant digit to least significant digit, in this case:

```
00 00 00 01
```

Intel processors use Little-Endian format and stores the numbers in reverse order with the least significant digit first, namely:

01 00 00 00

Figure 4-2 shows how the bytes in integers are copied into memory in both Little- and Big-Endian formats. Notice how the bytes end up in the reverse order to each other.

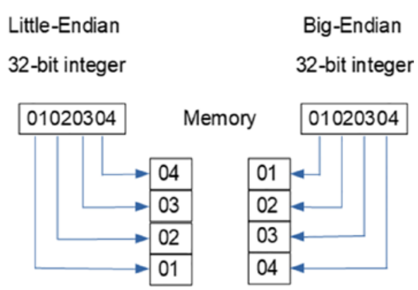

Figure 4-2. *How integers are stored in memory in Little- vs. Big-Endian format*

About Bi-Endian

The ARM CPU is called **Bi-Endian** because it can do either. There is a program status flag that says which endianness to use. By default, the RP2040 SDK uses Little-Endian like Intel processors.

Pros of Little-Endian

The advantage of the Little-Endian format is that it makes it easy to change the size of integers, without requiring any address arithmetic. If you want to convert a 4-byte integer to a 1-byte integer, you load the first byte, assuming the integer is in the range of 0–255 and the other three bytes are zero.

For example, if memory contains the 4 byte or word for 1, in Little-Endian, the memory contains

01 00 00 00

If we want the 1-byte representation of this number, we take the first byte; for the 16-bit representation, we take the first two bytes. The key point is that the memory address we use is the same in all cases, saving us an instruction cycle to adjust it.

When we are in the debugger, we will see more representations, and these will be pointed out again as they occur.

Cons of Little-Endian

Even though the RP2040 SDK uses Little-Endian, many protocols like TCP/IP used on the Internet use Big-Endian and so require a transformation when moving data from the RP2040 to the outside world. The other con is that the bytes are reversed to what a human is expecting, and this can lead to confusion when debugging.

How to Shift and Rotate Registers

We have 16 32-bit registers, and much of programming consists of manipulating the bits in these registers. Two extremely useful bit manipulations are shifting and rotating. Mathematically shifting all the bits left one spot is the same as multiplying by two, and generally shifting **n** bits is equivalent to multiplying by $\mathbf{2^n}$. Conversely, shifting bits to the right by **n** bits is equivalent to dividing by $\mathbf{2^n}$.

For example, consider shifting the number 3 left by 4 bits:

0000 0011 (the binary representation of the number 3)

Shift the bits left by 4 bits, and we get

```
0011 0000
```

which is

$0x30 = 3 * 16 = 3 * 2^4$

Now if we shift 0x30 right by 4 bits, we undo what we just did and see how it is equivalent to dividing by 2^4.

About Carry Flag

In the **CPSR**, there is a bit for **carry**. This is normally used to perform addition on larger numbers. If you add two 32-bit numbers and the result is larger than 32 bits, the carry flag is set. We'll see how to use this when we look at addition in detail later in this chapter. When we shift and rotate, it turns out to be useful to include the carry flag. This means we can do conditional logic based on the last bit shifted out of the register.

Basics of Shifting and Rotating

We have five cases to cover:

1. Logical shift left
2. Logical shift right
3. Arithmetic shift right
4. Rotate right
5. Rotate right extend

Logical Shift Left

This is quite straightforward, as we shift the bits left by the indicated number of places and zeros come in from the right. The last bit shifted out ends up in the carry flag.

Logical Shift Right

Equally easy as shifting the bits left, as we shift the bits right, zeros come in from the left, and the last bit shifted out ends up in the carry flag.

Arithmetic Shift Right

The problem with logical shift right is if it is a negative number with a zero coming in from the left, suddenly the number turns positive. If we want to preserve the sign bit, we use arithmetic shift right instead. This makes a 1 come in from the left if the number is negative and a 0 if it is positive. This is the correct form if you are shifting signed integers.

Rotate Right

Rotating is like shifting, except the bits don't go off the end—instead they wrap around and reappear from the other side. In this instance, rotate right shifts right, but the bits that leave on the right will reappear on the left.

Rotate Right Extend

Rotate right extend behaves like rotate right, except that it treats the register as a 33-bit register, where the carry flag is the 33rd bit and is to the right of bit 0. This type of rotation is limited to moving 1 bit at a time; therefore, the number of bits is not specified on the instruction.

How to Use MOV

In this section, we are going to learn the two forms of the **MOV** instruction:

1. MOV RD, #imm8
2. MOV RD, RS

Move Immediate

The first case is move immediate, and we've seen examples of this, putting a small number into a register. Here the immediate value can be any 8-bit quantity, and it will be placed in the lower eight bits of the specified register. This form of the **MOV** instruction is as simple as it gets; therefore, we will use it frequently. For example:

```
MOV R2, #3 @ Move 3 into register R2
```

Note Remember from Chapter 2 that most instructions encode registers as only 3 bits. When an instruction does this, then only the low registers **R0–R7** are valid, and that is the case for using the move immediate command.

Moving Data from One Register to Another Using Register MOV

In the second case, we have a version that moves one register into another. This is actually two separate instructions, one that moves between two low registers (**R0-R7**) while setting the **CPSR**. The second instruction moves between any registers but doesn't set the **CPSR**. This is one of the few instructions that allow us to access the high registers **R8–R15**.

Note Remember that **R12–R15** are special, and changing these will have side effects. **R12** is the intraprocedure call scratch register (**IP**), **R13** is the stack pointer (**SP**), **R14** is the link register (**LR**), and **R15** is the program counter (**PC**). If you move a value to **R15**, it will cause execution to jump to that location. We'll study how to properly use these registers in later chapters, so avoid them for now.

Here are some examples:

```
MOV R1, R2
MOVS R1, R2     @ the S explicitly states we want the first
                  version.
MOV R9, R3
MOV SP, R10     @ SP = R13
MOV PC, R11     @ PC = R15
```

We can now put small 8-bit values in a register, so let's start doing some arithmetic.

ADD/ADC

Let's start with addition. The instructions we'll cover are

1. ADD Rd, Rn, #imm3
2. ADD Rd, Rd, #imm8
3. ADD Rd, Rm, Rn
4. ADD Rd, Rd, Rm
5. ADD SP, SP, #imm7
6. ADD Rd, SP, #imm8
7. ADC Rd, Rd, Rm

These instructions all add their second and third parameters and put the result in their first parameter **Register Destination (Rd)**. A few notes on these instructions are

- Number 4, "ADD Rd, Rd, Rm", is the only one that allows any register (**R0–R15**) to be specified; since there are only two registers, a couple of extra bits are available.
- Except for number 4 and where **SP** is explicitly used, all the registers are low registers (**R0–R7**).
- All the immediate operands are positive integers.
- Numbers 5 and 6 are special instructions for dealing with the stack register. We'll see why these are necessary in Chapter 7.
- Only the instructions that deal with the low registers set the carry flag in the **CPSR**.
- The stack pointer must point to a word boundary, so any address in SP must be divisible by 4. As a result, only multiples of 4 are allowed in the immediate value, allowing it to be four times larger than expected.

Some examples are

```
ADD R4, R2, #7       @ this immediate allows 3 bits, so values 0-7
ADD R4, R4, #255     @ this one allows 8-bits, so 0-255
ADD R4, #255         @ alternate for for R4 = R4 + 255
ADD R10, R10, R13    @ The one instruction to allow high registers
ADD R10, R13         @ if one source register is the
                       destination, it can be omitted
ADD SP, #508         @ shouldn't do this without matching
                       subtraction
ADD R4, SP, #1020    @ 8-bit immediate so 0-1020 valid in steps
                       of 4
```


Add with Carry

The remaining instruction is Add with Carry (**ADC)**. This will be our first use of the **CPSR**.

Think back to how we learned to add numbers:

```
 17
+78
 95
```

1. We first add 7 + 8 and get 15.
2. We put 5 in our sum and carry the 1 to the tens column.
3. Now we add 1 + 7 + the carry from the ones column, so we add 1 + 7 + 1 and get 9 for the tens column.

This is the idea behind the carry flag. When an addition overflows, it sets the carry flag, so we can include that in the sum of the next part.

Note A carry is always 0 or 1, so we only need a 1-bit flag for this.

The ARM processor adds 32 bits at a time, so we only need the carry flag if we are dealing with numbers where the sum is larger than will fit into 32 bits. It turns out that we can use the carry flag to easily add 64-bit or larger numbers.

The carry flag is a bit in the **CPSR**; we'll look at the **CPSR** in more detail in Chapter 5. If the result of an addition is too large, then the carry flag is set to 1; otherwise, it is set to 0.

To add two 64-bit integers, use two 32-bit registers to hold each number. This example uses registers **R2** and **R3** for the first number, **R4** and **R5** for the second, and then **R0** and **R1** for the result. The code is

```
ADD R1, R3, R5   @ Lower order word
ADC R2, R4       @ Higher order word
MOV R0, R2       @ Move the result to where we want it
```

The first **ADD** adds the lower-order 32 bits and sets the carry flag, if needed. It might set other flags in the **CPSR**, but we'll worry about those later. The second instruction, **ADC**, adds the higher-order words, plus the carry flag.

Note **ADC** only takes two registers, so the sum overwrote our original number in **R2** which we moved into **R0** in the next instruction. If we still needed the original value of **R2**, it should be saved to another register first.

The nice thing here is that although we are in 32-bit mode, we can still do a 64-bit addition in only two clock cycles (three if we count the **MOV**).

SUB/SBC

Subtraction is the inverse of addition. We have

1. SUB Rd, Rn, Rm
2. SUB Rd, Rn, #imm3
3. SUB Rd, Rd, #imm8
4. SBC Rd, Rd, Rn
5. SUB SP, SP, #imm7
6. NEG Rd, Rn

The operands are the same as those for addition; only now we are calculating **Rn** – **Rm**. The carry flag is used to indicate when a borrow is necessary. **SUB** will clear the carry flag if the result is negative and set it if it's positive. **SBC** then subtracts one if the carry flag is clear.

NEG will negate a number: **Rd** = -**Rn**.

Shifting and Rotating

Here are the instructions for shifting and rotating the bits in a register:

1. LSL Rd, Rm, #shift5
2. LSL Rd, Rd, Rs
3. LSR Rd, Rm, #shift5
4. LSR Rd, Rd, Rs
5. ASR Rd, Rm, #shift5
6. ASR Rd, Rd, Rs
7. ROR Rd, Rd, Rs

These operations are logical shift left (**LSL**), logical shift right (**LSR**), arithmetic shift right (**ASR**), and rotate right (**ROR**). A few notes about these instructions

- The immediate value 5 bits gives values 0–31, sufficient for a 32-bit register.
- This set of instructions only operates on the low registers (**R0–R7**).
- The instructions that take three registers as operands can only operate in place (first and second operands must be the same and thus can be omitted).

Some examples:

```
LSL R1, R1, #2    @ Shift register R1 left 2 bits (multiply by 4)
LSL R1, #2        @ Shorter form if the registers are the same
LSR R1, R2, #8    @ Shift R2 right by one bytes and place the
                    result in R1
LSR R1, R3        @ Shift R1 right by the value in R3
ASR R1, #8        @ Arithmetic shift R1 right by one byte
ROR R1, R3        @ Rotate R1 right by value of R3
```

We've introduced quite a few instructions in this chapter; let's put a few of them together to load a 32-bit register.

Loading All 32 Bits of a Register

So far, we've seen how to load 8 bits with an immediate operation; but with MOV combined with shifting and adding, we can load all the bits. For example, let's load **R0** with the value 0x12345678. Our approach will be to do it 8 bits at a time. We will load 8 bits, shift it into position, and then add it in. Listing 4-1 contains the code for this.

Listing 4-1. Loading All 32 Bits on a Register

```
@ Initialize R0 with the leftmost byte
      MOV R0, #0x12      @ load the first 8-bits
      LSL R0, #24        @ shift it left 24 bits into place
@ Load the next byte into R1
      MOV R1, #0x34      @ load the second byte
      LSL R1, #16        @ shift it into place
      ADD R0, R2         @ add it into R1
@ repeat for the third byte
      MOV R1, #0x56      @ load the third byte
      LSL R1, #8         @ shit it into place
      ADD R0, R1         @ add it to the sum
```



```
@ for the last byte no shift required
      MOV R1, #0x78      @ load the fourth bytes
      ADD R0, R1
```

That was a bit of work and demonstrates that working with a small set of instructions can create quite a few program statements, but remember each instruction is only 16 bits in size. In Chapter 6, we'll learn how to load registers from memory, which is less code, but we will see cases later where tricks like this result in quick ways to load registers (especially if there are zeros in the middle). Next is an example containing all these instructions.

MOV/ADD/Shift Example

If we combine all the small code samples in this chapter with our 32-bit register loading and 64-bit addition, we get Listing 4-2. This program ensures the registers are initialized and provides comments of what the results should be. There is a label "after" after the call to stdio_init_all, which is a good place to set a breakpoint, and then single-step through the code. Use gdb's "**i r**" command frequently to check the values of the registers. At the end, the program prints out the 64-bit sum from the addition.

1. Create a new project folder.
2. Create a file called "movaddsubshift.S" containing Listing 4-2 in that folder.

Listing 4-2. Examples of the MOV, ADD, and Shift Instructions Along with 64-Bit Addition

```
@
@ Examples of the MOV/ADD/SUB/Shift instructions.
@

.thumb_func             @ Necessary because sdk uses BLX
.global main            @ Provide program starting address
                          to linker

main:   BL stdio_init_all   @ initialize uart or usb

after:  MOV R2, #3          @ Move 3 into register R2
        MOV R1, R2          @ R1 is now also 3
        MOVS R1, R2         @ the S explicitly states we want
                              the first version.
        MOV R9, R2          @ R9 now is 3

@ we shouldn't play with SP or PC until we know what we're
  doing.
        @ MOV SP, R10       @ SP = R13
        @ MOV PC, R11       @ PC = R15

        ADD R4, R2, #7      @ this immediate allows 3 bits, so
                              values 0-7
@ R4 is now 10 (3 + 7)
        ADD R4, R4, #255    @ this one allows 8-bits, so 0-255
@ R4 is now 265 (10 + 255)
        ADD R4, #255        @ alternate for for R4 = R4 + 255
@ R4 is now 520(265 + 255)
        MOV R7, #23         @ Can't load high registers with
                              immediate
        MOV R11, R7         @ So load R7 and move it
        MOV R7, #54
```



```
        MOV R10, R7         @ if one source register is the
                              destination, it can be omitted
        ADD R10, R10, R11   @ The one instruction to allow
                              high registers
@ R10 is now 77 (23 + 54)
        ADD SP, SP, #508    @ shouldn't do this without
                              matching subtraction
        SUB SP, SP, #508    @ Undo the damage.
        ADD R4, SP, #1020   @ 8-bit immediate but multiples of
                              4 so 0-1020 valid
@ need to check R4 in the debugger since it depends on the
  value of SP
@ when I ran I got 0x200423fc but if SDK changes this could
  change.
@ Repeat the above shifts using the Assembler mnemonics.

        MOV R3, #8          @ will use this to shift or rotate
                              1-byte
        MOV R2, #0xFF       @ R2 = 255
        MOV R1, #4          @ R1 = 4
        LSL R1, R1, #2      @ Shift register R1 left 2 bits
                              (multiply by 4)
        LSL R1, #2          @ Shorter form if the registers
                              are the same
        LSR R1, R2, #8      @ Shift R2 right by one bytes and
                              place the result in R1
        LSR R1, R3          @ Shift R1 right by the value in R3
        ASR R1, #8          @ Arithmetic shift R1 right by one
                              byte
        ROR R1, R3          @ Rotate R1 right by value of R3
```

```
@ Load 0x12345678 into R3
@ Initialize R3 with the leftmost byte
        MOV R3, #0x12       @ load the first 8-bits
        LSL R3, #24         @ shift it left 24 bits into place
@ Load the next byte into R1
        MOV R1, #0x34       @ load the second byte
        LSL R1, #16         @ shift it into place
        ADD R3, R1          @ add it into R1
@ repeat for the third byte
        MOV R1, #0x56       @ load the third byte
        LSL R1, #8          @ shit it into place
        ADD R3, R1          @ add it to the sum
@ for the last byte no shift required
        MOV R1, #0x78       @ load the fourth bytes
        ADD R3, R1

@ Other registers for our upcoming 64-bit addition
        MOV R2, #0x12
        MOV R4, #0x54
        MOV R5, #0xf0
        LSL R5, #24         @ shift f0 over to the high byte

@ 64-bit Addition (rigged to cause a carry)
@ Do sum:
@       R2 R3   0x12 0x12345678
@       R4 R5   0x54 0xF0000000
@       -----  -----------------
@       R0 R1   0x67 0x02345678

        ADD R1, R3, R5      @ Lower order word
        ADC R2, R4          @ Higher order word
        MOV R0, R2          @ Move the result to where we want it
```



```
@ Save R0, R1 since printf will overwrite them
        MOV R6, R0          @ R6 = R0
        MOV R7, R1          @ R7 = R1

@ print out the sum
loop:   MOV R1, R6          @ R1 is param2
        MOV R2, R7          @ R2 is param3
        LDR R0, =sumstr     @ load address of sumstr to param1
        BL  printf          @ call printf
        B   loop            @ loop in case uart monitoring not
                              started
.data
        .align  4           @ necessary alignment
sumstr: .asciz   "The sum is %x %x\n"
```

Listing 4-3 contains the CMakeLists.txt file needed to build this sample. Remember you need to copy pico_sdk_import.cmake to the project folder.

Listing 4-3. The CMakeLists.txt File for Our Sample

```
cmake_minimum_required(VERSION 3.13)

include(pico_sdk_import.cmake)
project(MovAddSub C CXX ASM)

set(CMAKE_C_STANDARD 11)
set(CMAKE_CXX_STANDARD 17)

pico_sdk_init()

include_directories(${CMAKE_SOURCE_DIR})

add_executable(MovAddSub
  movaddsubshift.S
)
```

```
pico_enable_stdio_uart(MovAddSub 1)
pico_enable_stdio_usb(MovAddSub 0)

pico_add_extra_outputs(MovAddSub)

target_link_libraries(MovAddSub pico_stdlib)
```

After you build the program, have a look at MovAddSub.dis. The program consists of 47 16-bit instructions and two 32-bit instructions (the two **BL** instructions). This means the program contains 102 bytes of code. Even though it takes quite a few instructions to get meaningful work done, the end program ends up being extremely compact.

The program avoided making changes to registers **R12–R15**, because if we change **R15** (the program counter), the program will jump to the address we set, which in this case we don't want. Registers **R12–R14** are used when functions are called, and if we change these, the call to printf won't work. We'll see how to change **R15** in Chapter 5. We'll see how to use **R12–R14** in Chapter 7.

Summary

In this chapter, we learned how negative integers are represented in a computer. We went on to discuss Big- vs. Little-Endian byte ordering and then introduced the concept of shifting and rotating the bits in a register.

Next, we viewed in detail the **MOV** instruction that allows us to move data around the CPU registers or load constants from the **MOV** instruction into a register.

We covered the **ADD** and **ADC** instructions and discussed how to add both 32- and 64-bit numbers. We briefly introduced the **SUB** and **SBC** instructions. Finally, we offered the various shift and rotation instructions.

We then put the instructions together to load all 32 bits of a register and incorporated them into an example program to add two 64-bit integers.

In Chapter 5, we will conditionally execute code and learn to branch and loop the program, which are the core building blocks of programming logic.

Exercises

4-1. Compute the 8-bit two's complement for -79 and -23.

4-2. What are the negative decimal numbers represented by the bytes 0xF2 and 0x83?

4-3. Manually write out the bytes in the Little-Endian representation of 0x12345678.

4-4. Manually write out the bytes for 0x23 shifted left by 3 bits.

4-5. Manually write out the bytes for 0x4300 right shifted by 5 bits.

4-6. Code a program to add two 96-bit numbers. How will you manage the limited number of registers?

4-7. Code a program that performs 64-bit subtraction. Convince yourself that the way it sets and interprets the carry flag is what you need in this situation. Use it to reverse the operations from the 64-bit addition in Listing 4-2.

CHAPTER 5

How to Control Program Flow

Now we know a handful of Assembly Language instructions and can execute them linearly one after the other. We built programs and debugged them. In this chapter, we'll make our programs more interesting by using conditional logic—**if/then/else** statements, in high-level language. We will also introduce loops—**for** and **while** statements, in high-level languages. With these instructions in hand, we will have all the basics for coding program logic.

Unconditional Branch

The simplest branch instruction is

```
B label
```

that is an unconditional branch to a label. The label is interpreted as an offset from the current **PC** register and has 11 bits in the instruction, allowing a range of -2048 to 2046. 2^{11} is 2048, but since instructions must be on even addresses, this offset is multiplied by 2. This instruction is like a **goto** statement in some high-level languages.

S. Smith, *RP2040 Assembly Language Programming*,
https://doi.org/10.1007/978-1-4842-7753-9_5

About the CPSR

We've mentioned the Current Program Status Register (**CPSR**) several times without really looking at what it contains. We talked about the carry flag when we looked at the **ADD**/**ADC** instructions. In this section, we will look at a few more of the flags in the **CPSR**.

We'll start by listing all the flags it contains, though many of them won't be discussed until later chapters. In this chapter, we are interested in the group of condition code bits that tell us things about what happens when an instruction executes (Figure 5-1).

31	30	29	28	27	26-0
N	Z	C	V	Q	Reserved

Figure 5-1. *The bits in the CPSR*

The condition flags are

- **N**egative: **N** is 1 if the signed value is negative and cleared if the result is positive or 0.
- **Z**ero: This flag is set if the result is 0; this usually denotes an equal result from a comparison. If the result is nonzero, this flag is cleared.
- **C**arry: For addition-type operations, this flag is set if the result produces an overflow. For subtraction-type operations, this flag is set if the result requires a borrow. Also, it's used in shifting to hold the last bit that is shifted out.
- O**V**erflow: For addition and subtraction, this flag is set if a signed overflow occurred. **Note**: Some instructions may specifically set o**V**erflow to flag an error condition.
- **Q**: This flag is set to indicate underflow and/or saturation.

Branch on Condition

The branch instruction, at the beginning of this chapter, can take a modifier that instructs it to only branch if a certain condition flag in the **CPSR** is set or clear.

The general form of the branch instructions is

```
B{condition} label
```

where {condition} is taken from Table 5-1.

Table 5-1. *Condition Codes for the Branch Instruction*

{condition}	Flags	Meaning
EQ	Z set	Equal
NE	Z clear	Not equal
CS or **HS**	C set	Higher or same (unsigned >=)
CC or **LO**	C clear	Lower (unsigned <)
MI	N set	Negative
PL	N clear	Positive or zero
VS	V set	Overflow
VC	V clear	No overflow
HI	C set and Z clear	Higher (unsigned >)
LS	C clear and Z set	Lower or same (unsigned <=)
GE	N and V the same	Signed >=
LT	N and V differ	Signed <
GT	Z clear, N and V the same	Signed >
LE	Z set, N and V differ	Signed <=
AL	Any	Always (same as no suffix)

For example:

```
BEQ main
```

will branch to main if the **Z** flag is set. This seems a bit strange—why isn't the instruction **BZ** for branch on zero? What is equal here? To answer these questions, we need to look at the **CMP** instruction.

About the CMP Instruction

There are two forms of the **CMP** instruction:

1. CMP Rn, Rm
2. CMP Rn, #imm8

This instruction compares the contents of register **Rn** with the second operand by subtracting the second operand from **Rn** and updating the status flags accordingly. It behaves exactly like the **SUB** instruction, except that it only updates the status flags and discards the result. For example, to do a branch only if register **R4** is 45, we might code

```
CMP R4, #45
BEQ main
```

In this context, we see how the mnemonic **BEQ** makes sense: since **CMP** subtracts 45 from **R4**, the result is zero if they are equal, and the **Z** flag will be set. If you go back to Table 5-1 and consider the condition codes in this context, then they make sense.

Note Both registers can be low registers (**R0–R7**), or one register can be high (**R8–R15**) and one register low (**R0–R7**). Both registers cannot be high registers.

Loops

With branch and comparison instructions in hand, let's look at constructing some loops modelled on what we find in high-level programming languages.

FOR Loops

Suppose we want to do the Basic FOR loop:

```
FOR I = 1 to 10
    ... some statements...
NEXT I
```

We can implement this as shown in Listing 5-1.

Listing 5-1. Basic FOR Loop

```
      MOV R2, #1     @ R2 holds I
loop: @ body of the loop goes here.

      @ Most of the logic is at the end
      ADD R2, #1     @ I = I + 1
      CMP R2, #10
      BLE loop       @ IF I <= 10 goto loop
```

If we did this by counting down

```
FOR I = 10 TO 1 STEP -1
    ... some statements...
NEXT I
```

we can implement this as shown in Listing 5-2.

Listing 5-2. Reverse FOR Loop

```
      MOV R2, #10     @R2 holds I
loop: @ body of the loop goes here.

      @ The CMP is redundant since we
      @ are doing SUB.
      SUB R2, #1      @ I = I -1
      BNE loop        @ branch until I = 0
```

Here, we save an instruction, since with the **SUB** instruction, we don't need the **CMP** instruction.

While Loops

Let's code:

```
WHILE X < 5
    ... other statements ....
END WHILE
```

Initializing the variables and changing the variables aren't part of the **while** statement. These are separate statements that appear before and in the body of the loop. In Assembly, we might code as shown in Listing 5-3.

Listing 5-3. While Loop

```
      @ R4 is X and has been initialized
loop: CMP R4, #5
      BGE loopdone
          ... other statements in the loop body ...
      B   loop
loopdone: @program continues
```

Note A while loop only executes if the statement is initially true, so there is no guarantee that the loop body will ever be executed.

If/Then/Else

In this section, we'll look at coding:

```
IF <expression> THEN
    ... statements ...
ELSE
    ... statements ...
END IF
```

In Assembly Language, we need to evaluate <expression> and have the result end up in a register that we can compare. For now, we'll assume that <expression> is simply of the form

```
register comparison immediate-constant
```

In this way, we can evaluate it with a single **CMP** instruction. For example, suppose we want to code

```
IF R5 < 10 THEN
    .... if statements ...
ELSE
    ... else statements ...
END IF
```

We can code this as Listing 5-4.

Listing 5-4. If/Then/Else Statement

```
CMP R5, #10
      BGE elseclause

      ... if statements ...

      B endif
elseclause:

      ... else statements ...

endif:    @ continue on after the /then/else ...
```

This is fairly simple, but it is still worth putting in comments to be clear which statements are part of the if/then/else and which statements are in the body of the if or else blocks.

Tip Adding a blank line can make the code much more readable.

Logical Operators

For our upcoming sample program, we need to start manipulating the bits in the registers. The ARM Cortex-M0+'s logical operators provide several tools for us to do this:

1. AND Rd, Rd, Rm
2. EOR Rd, Rd, Rm
3. ORR Rd, Rd, Rm
4. BIC Rd, Rd, Rm
5. MVN Rd, Rm
6. TST Rn, Rm

These operate on each bit of the registers separately. A couple of notes:

- All of these instructions only operate on the low registers (**R0–R7**).
- For all the instructions where the first two operands are the same, they can be shortened to specify two registers.

Figure 5-2 shows what each logical operation does to each combination of input bits.

X	Y	X AND Y	X EOR Y	X ORR Y	X BIC Y
0	0	0	0	0	0
0	1	0	1	1	0
1	0	0	1	1	1
1	1	1	0	1	0

Figure 5-2. *What each logical operator does with each pair of bits*

AND

AND performs a bitwise logical and operation between bits in **Rd** and **Rm**, putting the result in **Rd**. Remember that logical and is true (1) if both arguments are true (1) and false (0) otherwise.

Let's use **AND** to mask off a byte of information. Suppose we only want the high-order byte of a register (Listing 5-5).

Listing 5-5. Using AND to Mask a Byte of Information

```
@ mask off the high-order byte
MOV R5, #0xFF
LSL R5, #24    @ R5 = 0xFF000000
AND R6, R5
```


This code will preserve the high-order byte while zeroing out the other three bytes. It takes us two instructions to load the mask: one to load 0xFF and then an **LSL** instruction to shift it into the correct position.

EOR

EOR performs a bitwise exclusive or operation between bits in **Rd** and **Rm**, putting the result in **Rd**. Remember that exclusive or is true (1) if exactly one argument is true (1) and false (0) otherwise.

ORR

ORR performs a bitwise logical or operation between bits in **Rd** and **Rm**, putting the result in **Rd**. Remember that logical or is true (1) if one or both arguments are true (1) and false (0) if both arguments are false (0), for example:

```
MOV R5, #0xFF     @ Load he second argument
ORR R6, R5        @ Perform R6 = R6 or R5
```

This sets the low-order byte of **R6** to all 1 bits (0xFF) while leaving the three other bytes unaffected.

BIC

BIC (Bit Clear) performs **Rd** and not **Rm**. The reason is that if the bit in **Rm** is 1, then the matching bit in **Rd** will be set to 0. If the bit in **Rm** is 0, then the corresponding bit in **Rd** will be unaffected.

MVN

MVN (Move Not) performs a bitwise not operation on each bit or **Rm** and places the result in **Rd**. This calculated the one's complement of **Rd**.

TST

TST (And Test) performs an **AND** operation between **Rn** and **Rm**, setting the condition flags and then discarding the result. This is similar to the **CMP** instruction, but using **AND** instead of **SUB**. For example:

```
MOV R5, #0xFF   @ load R5 with 0xFF
TST R6, R5      @ set R6 = R5 and R6
BNE lowbits     @ if non-zero then there're low order bits
```

Design Patterns

When writing Assembly Language code, there is a great temptation to be creative. For instance, we could do a loop ten times by setting the tenth bit in a register and then shifting it right until the register is zero. This works, but it makes reading your program difficult. If you leave your program and come back to it at a later date, you will be scratching your head as to what the program does.

Design patterns are typical solutions to common programming patterns. If you adopt a few standard design patterns on how to perform loops and other programming constructs, it will make reading your programs much easier.

Design patterns make your programming more productive, since you can just use an example from a collection of tried-and-true patterns for most situations.

Tip In Assembly, make sure you document which design pattern you are using, along with documenting the registers used.

Therefore, we implemented loops and if/then/else in the pattern of a high-level language. If we do this, it makes our programs more reliable and quicker to write. Later, we'll look at how to use the macro facility in the GNU Assembler to help with this.

Converting Integers to ASCII

As a first example of a loop, let's convert a 32-bit register to ASCII. In our HelloWorld program in Chapter 2, we used the RP2040 SDK's printf function to output our "Hello World!" string. In this program, we will convert the hex digits in the register to ASCII characters digit by digit. ASCII is one way that computers represent all the letters, numbers, and symbols that we read as numbers that a computer can process. For instance:

- A is represented by 65.
- B is represented by 66.
- 0 is represented by 48.
- 1 is represented by 49, and so on.

The key point is that the letters A to Z are contiguous as are the numbers 0 to 9. See Appendix A for all 255 characters.

Note For a single ASCII character that fits in 1 byte, enclose it in single quotes, for example, 'A'. If the ASCII characters are going to comprise a string, use double quotes, for example, "Hello World!".

Here is some high-level language pseudocode for what we will implement in Assembly Language (Listing 5-6).

Listing 5-6. Pseudocode to Convert a Register to ASCII

```
outstr = memory where we want the string + 9
@ (string is form 0x12345678 and we want
@ the last character)
FOR R5 = 8 TO 1 STEP -1
       digit = R4 AND 0xf
       IF digit < 10 THEN
             asciichar = digit + '0'
       ELSE
             asciichar = digit + 'A' - 10
       END IF
       *outstr = asciichar
       outstr = outstr - 1
NEXT R5
```

Listing 5-7 is the Assembly Language program to implement this. It uses what we learned about loops, if/else, and logical statements. Create a project folder for this along with a CMakeLists.txt as we have done in previous samples.

Listing 5-7. Printing a Register in ASCII

```
@ Example to convert contents of register to ASCII
@
@ R0-R1 - parameters printf
@ R1 - is also address of byte we are writing
@ R4 - register to print
@ R5 - loop index
@ R6 - current character
@ R7 - temp register
```



```
.thumb_func                @ Necessary because sdk uses BLX
.global main               @ Provide program starting address
                             to linker

main: BL stdio_init_all    @ initialize uart or usb

printexample:
       @ Load R4 with 0x12AB
       MOV R4, #0x12       @ number to print
       LSL R4, #8
       MOV R7, #0xAB
       ADD R4, R7
       LDR R1, =hexstr     @ start of string
       ADD R1, #9          @ start at least sig digit
@ The loop is FOR r5 = 8 TO 1 STEP -1
       MOV R5, #8          @ 8 digits to print
loop4: MOV R6, R4
       MOV R7, #0xf
       AND R6, R7          @ mask of least sig digit
@ If R6 >= 10 then goto letter
       CMP R6, #10         @ is 0-9 or A-F
       BGE letter
@ Else it's a number so convert to an ASCII digit
       ADD R6, #'0'
       B cont              @ goto to end if
letter: @ handle the digits A to F
       ADD R6, #('A'-10)
cont:    @ end if
       STRB R6, [R1]       @ store ascii digit
       SUB  R1, #1         @ decrement address for next digit
       LSR  R4, #4         @ shift off the digit we just
                             processed
```

```
        @ next R5
        SUB R5, #1          @ step R5 by -2
        BNE loop4           @ another for loop if not done

repeat:
        LDR R0, =printstr
        LDR R1, =hexstr @ string to print
        BL  printf
        B   repeat

.align 4
.data
hexstr:      .asciz  "0x12345678"
printstr:    .asciz  "Register = %s\n"
```

The best way to understand this program is to single-step through it in **gdb** and watch how it is using the registers and updating memory. Remember from Chapter 1 that you need to create a debug build with the UART set for printing, have the updated .**gdbinit** in place, and run **openocd** via the **ocdg** script.

Make sure you understand why

```
MOV R7, #0xf
AND R6, R7 @ mask of least sig digit
```

masks off the low-order digit; if not, review the "AND" section on logical operators.

Since **AND** requires both operands to be 1 in order to result in 1, and'ing something with 1s (like 0xf) keeps the other operator as is, whereas and'ing something with 0s always makes the result 0.

In our loop, we shift **R4** 4 bits right with

```
LSR R4, #4
```

This shifts the next digit into position for processing in the next iteration.

Note This is destructive to **R4**, and you will lose your original number during this algorithm.

We've already discussed most of the elements present in this program, but there are a couple of new elements; they are demonstrated in the following.

Using Expressions in Immediate Constants

```
ADD R6, #('A'-10)
```

This demonstrates a couple of new tricks from the GNU Assembler:

1. We can include ASCII characters in immediate operands by putting them in single quotes.
2. We can place simple expressions in the immediate operands. The GNU Assembler translates 'A' to 65, subtracts 10 to get 55, and uses that as Operand2.

This makes the program more readable, since we can see our intent, rather than if we had just coded 55 here. There is no penalty to the program in doing this, since the work is done when we assemble the program, not when we run it.

Storing a Register to Memory

```
STRB R6, [R1]
```

The Store Byte (**STRB**) instruction saves the low-order byte of the first register into the memory location contained in **R1**. The syntax [**R1**] is to make clear that we are using memory indirection and not just putting the byte into register **R1**. This is to make the program more readable, so we don't confuse this operation with a corresponding **MOV** instruction.

Accessing data in memory is the topic of Chapter 5, where we will go into far greater detail. The way we are storing the byte could be made more efficient, and we'll look at that then.

Why Not Print in Decimal?

In this example program, we easily convert to a hex string because using **AND** 0xf is equivalent to getting the remainder when dividing by 16. Similarly, shifting the register right 4 bits is equivalent to dividing by 16. If we wanted to convert to a decimal, base 10, string, then we would need to be able to get the remainder from dividing by 10 and later divide by 10.

So far, we haven't seen a divide instruction. This places converting to decimal beyond the scope of this chapter. We could write a loop to implement the long division algorithm we learned in elementary school, but instead we will defer division until Chapter 13.

Performance of Branch Instructions

In Chapter 2, we mentioned that the ARM Cortex-M0+ instruction set is executed in an instruction pipeline. Individually, an instruction requires three clock cycles to execute, one for each of the following:

1. Load the instruction from memory to the CPU.
2. Decode the instruction.
3. Execute the instruction.

However, the CPU works on three instructions at once, each at a different step, so on average, we execute one instruction every clock cycle. But what happens when we branch?

When we execute the branch, we've already decoded the next instruction and loaded the instruction 2 ahead. When we branch, we throw this work away and start over. We see this in the ARM documentation

that most branch instructions take two clock cycles to execute, whereas most other instructions only take one. For a conditional branch, there is no penalty if we don't take the branch and a **BL** instruction takes an extra cycle.

If you put a lot of branches in your code, you suffer a performance penalty. Another problem is that if you program with a lot of branches, this leads to spaghetti code—meaning all the lines of code are tangled together like a pot of spaghetti, which is understandably quite hard to maintain.

When I first learned to program in high school and my undergraduate years before structured programming was available, I used the BASIC and Fortran programming languages to write complex code. I know firsthand that deciphering programs full of branches is a challenge.

Early high-level programming languages relied on the **goto** statement that led to hard-to-understand code; this led to the structured programming we see in modern high-level languages that don't need a **goto** statement. We can't entirely do away with branches, since ARM Cortex-M0 Assembly Language doesn't have structured programming constructs, but we need to structure our code along these lines to make it both more efficient and easier to read—another great use for design patterns.

Summary

In this chapter, we studied the key instructions for performing program logic with loops and **if** statements. These included the instructions for comparisons and conditional branching. We discussed several design patterns to code the common constructs from high-level programming languages in Assembly Language. We looked at the statements for logically working with the bits in a register. We examined how we could output the contents of a register in hexadecimal format.

In Chapter 6, we'll look at the details of how to load data to and from memory.

Exercises

5-1. Go through Table 5-1 of condition codes and ensure you understand why each one is named the way it is.

5-2. Create an Assembly Language framework to implement a SELECT/CASE construct. The format is

```
SELECT number
      CASE 1:
            << statements if number is 1 >>
      CASE 2:
            << statements if number is 2>>
      CASE ELSE:
            << statements if not any other case >>
END SELECT
```

5-3. Construct a DO/WHILE statement in Assembly Language. In this case, the loop always executes once before the condition is tested:

```
DO
    << statements in the loop >>
UNTIL condition
```

5-4. Modify the program in Listing 5-7 to print the hex representation of two registers assuming that combined they hold a 64-bit integer.

CHAPTER 6

Thanks for the Memories

In this chapter, we discuss the memory of the RP2040. So far, we've used memory only to hold Assembly Language instructions. Now, we will look in detail at how to define data in memory, then how to load memory into registers for processing, and how to write the results back to memory.

The ARM Cortex-M0+ uses a load-store architecture. This means that the instruction set is divided into two categories: one to load and store values from and to memory and the other to perform arithmetic and logical operations between the registers. We've spent most of our time looking at the arithmetic and logical operations. Let's look at the other category of load-store.

Memory addresses are 32 bits and instructions are 16 bits, so we encounter the same problems experienced in Chapter 4, where we used all sorts of tricks to load 32 bits into a register. In this chapter, we'll use these same tricks for loading addresses, along with several new ones. The goal is to load a 32-bit address in one instruction in as many cases as we can.

Before we load and build memory addresses, we need to define the contents of memory with the GNU Assembler.

S. Smith, *RP2040 Assembly Language Programming*,
https://doi.org/10.1007/978-1-4842-7753-9_6

How to Define Memory Contents

The GNU Assembler contains several directives to help define memory to use in your program. These appear in a **.data** section of your program. We'll look at examples and then summarize them in Table 6-1. Listing 6-1 shows how to define bytes, words, and ASCII strings.

Listing 6-1. Sample Memory Directives

```
label: .byte 74, 0112, 0b00101010, 0x4A, 0X4a, 'J', 'H' + 2
       .word 0x1234ABCD, -1434
       .ascii "Hello World\n"
```

The first line defines seven bytes all with the same value. We can define bytes in decimal, octal (base 8), binary, hex, or ASCII. Anywhere numbers are defined, we can use expressions that the Assembler will evaluate when it compiles our program.

We start most memory directives with a label, so we can access it from the code. The only exception is if we are defining a larger array of numbers that extends over several lines.

The **.byte** statement defines one or more bytes of memory. Listing 6-1 shows the various formats we can use for the contents of each byte, as follows:

- A decimal integer starts with a nonzero digit and contains decimal digits 0–9.
- An octal integer starts with zero and contains octal digits 0–7.
- A binary integer starts with 0b or 0B and contains binary digits 0–1.
- A hex integer starts with 0x or 0X and contains hex digit 0–F.
- A floating-point number starts with 0f or 0e, followed by a floating-point number.

Note Do not start decimal numbers with zero (0), since this indicates the constant is an octal (base 8) number.

The example then shows how to define a word and an ASCII string, as we saw in our HelloWorld program in Chapter 1. There are two prefix operators we can place in front of an integer:

- Negative (-) will take the two's complement of the integer.
- Complement (~) will take the one's complement of the integer.

For example:

```
.byte -0x45, -33, ~0b00111001
```

Table 6-1 lists the various data types we can define this way.

Table 6-1. *The List of Memory Definition Assembler Directives*

Directive	Description
.ascii	A string contained in double quotes
.asciz	A zero-byte terminated ASCII string
.byte	1 byte integers
.double	Double-precision floating-point values
.float	Floating-point values
.octa	16-byte integers
.quad	8-byte integers
.short	2-byte integers
.word	4-byte integers

If we want to define a larger set of memory, there are a couple of mechanisms to do this without having to list and count them all, such as

```
.fill repeat, size, value
```

This repeats a value of a given size, repeat times; for example:

```
zeros: .fill 10, 4, 0
```

creates a block of memory with ten 4-byte words all with a value of zero. The code

```
.rept count
...
.endr
```

repeats the statements between **.rept** and **.endr**, count times. This can surround any code in Assembly Language; for instance, you can make a loop by repeating the code count times; for example:

```
.rept 3
.byte 0, 1, 2
.endr
```

is translated to

```
.byte 0, 1, 2
.byte 0, 1, 2
.byte 0, 1, 2
```

In ASCII strings, we've seen the special character "\n" for a new line. There are a few more for common unprintable characters, as well as for double quotes in strings. The "\" is called an escape character, which is a metacharacter to define special cases. Table 6-2 lists the escape character sequences supported by the GNU Assembler.

Table 6-2. *ASCII Escape Character Sequence Codes*

Escape character sequence	Description
\b	Backspace (ASCII code 8)
\f	Formfeed (ASCII code 12)
\n	New line (ASCII code 10)
\r	Return (ASCII code 13)
\t	Tab (ASCII code 9)
\ddd	An Octal ASCII code (ex \123)
\xdd	A Hex ASCII code (ex \x4F)
****	The '\' character
\\"	The double quote character
\anything-else	Anything else

How to Align Data

These data directives put the data in memory contiguously byte by byte. However, ARM processors often require data to be aligned on word boundaries or by some other measure. We can instruct the Assembler to align the next piece of data with an **.align** directive. For instance, consider

```
.data
.byte 0x3F
.align 4
.word 0x12345678
```

The first byte is word aligned, but because it is only 1 byte, the next word of data will not be aligned. If we need it to be word aligned, then add the ".align 4" directive. This will result in three wasted bytes, but if this is a problem, you may need to rearrange your memory data.

ARM Cortex-M0+ Assembly Language instructions must be 16-bit aligned, so if data is inserted in the middle of some instructions, then add an **.align** directive before the instructions continue, or our program will crash when it is run. In the next section, we'll see that when data is loaded with **PC** relative addressing, these addresses must also be appropriately aligned. Usually, the Assembler gives an error when alignment is required, and throwing in an ".align 2" or ".align 4" directive is a quick fix.

How to Load a Register

In this section, we will look at the **LDR** instruction and its variations. We use **LDR** to both load an address into a register and to load the data pointed to by that address. There are methods to index through memory, as well as support for all the strategies to get as much as possible out of the 16-bit instructions. We'll go through the cases one by one, including

- Loading a memory address into a register
- Loading data from memory
- Indexing through memory

Note All the load and store instructions operate only on the low registers (**R0–R7**); the only exceptions are **PC** and **SP** relative addressing that explicitly use **PC** and **SP**.

We'll first look at how to load or create a memory address in a register.

How to Load a Register with an Address

To create a memory address in a register, we can either create it from scratch or base it on an address that is already in another register. First of all, we'll build the address directly.

How to Build the Address Directly

When you write a program under a modern operating system, like Linux, you can't just create a memory address; you have to ask the operating system to provide the address, and this takes into account virtual memory and memory protection. On a microcontroller, like the RP2040, there is no operating system, virtual memory, memory management, or memory protection. The memory map of the RP2040 is fixed and documented in the RP2040's SDK reference documentation. As a consequence, there are many situations where we know the address we want ahead of time and need to load it into a register to use. In the previous chapter, we learned how to load a 32-bit register with any value, and this will work in this situation. Fortunately, the addresses we want to deal with are often fairly simple, such as 0xd0000014, which is the memory address we write to for setting GPIO pins. Since most of the address is 0s, we can load it into a register with

```
MOV R2, #0xd0
LSL R2, R2, #24 @ becomes 0xd0000000
ADD R2, #0x14
```

Here, we took three 16-bit instructions to build the address into **R2** and didn't require any additional memory. Code like this can be tricky, so make sure you document it. Next, we'll look at a more straightforward way of building addresses using an existing memory address in the program counter (**PC**).

PC Relative Addressing

In Chapter 2, we introduced the **LDR** instruction to load the address of our "Hello World!" string. We needed to do this to pass the address of what to print to the RP2040 SDK's **printf** function. This is a simple and convenient example of **PC** relative addressing, since it doesn't involve any other registers. As long as you keep your data close to your code, it is painless, as when we looked at the disassembly of the **LDR** instruction:

```
LDR R0, =helloworld
```

was

```
ldr r0, [pc, #12]; (10000370 <loop+0xe>)
```

Here, we are writing an instruction to load the address of our "helloworld" string into **R0**. The Assembler knows the value of the program counter at this point, so it can provide an offset to the correct memory address. Therefore, it's called **PC** relative addressing. There is a bit more complexity to this, which we'll address soon.

The offset above has 8 bits in the instruction with a range of 0–255. To get a greater range, the target address has to be 32-bit aligned, which means the effective range is multiplied by four, so we get a range of 0–1020.

Note We can also do this relative to the stack pointer (**SP**); however, we'll examine the SP in detail in Chapter 7.

How to Load Data from Memory

In our HelloWorld program, we only needed the address to pass on to the **printf**, which is used to print our string. Generally, we like to use these addresses to load data into a register.

The simple form of **LDR** to load data given an address is

```
LDR{type} Rd, [Rm]
```

where type is one of the types listed in Table 6-3.

Table 6-3. *The Data Types for the Load/Store Instructions*

Type	Meaning
B	Unsigned byte
SB	Signed byte
H	Unsigned halfword (16 bits)
SH	Signed halfword (16 bits)
SW	Signed word (32 bits)
<none>	Unsigned word (32 bits)

Listing 6-2 demonstrates the two-step process to load a register. First, we load **R1** with the address of the data we want; then we use that register to indirectly load register **R2** with the actual data.

Listing 6-2. Loading an Address and Then the Value

```
@ load the address of mynumber into R1
LDR R1, =mynumber
@ load the word stored at mynumber into R2
LDR R2, [R1]
.data
mynumber: .WORD 0x1234ABCD
```

If you step through this in the debugger, you can watch it load 0x1234ABCD into **R2**.

Note The square bracket syntax represents indirect memory access. This means load the data stored at the address pointed to by **R1**, not move the contents of **R1** into **R2**.

When we encountered “LDR r0, [pc, #12]”, it looked like loading the address of pc+12, but we are actually loading the data stored at pc+12, which is why square brackets are used. This works since the Assembler placed the address we want at this location.

This works, but you might be dissatisfied that it took two instructions to load **R2** with our value from memory: one to load the address and then one to load the data. When programming a RISC processor, each instruction executes extremely quickly but performs only a small chunk of work. We can do a little better than this in some instances for read-only quantities.

Optimizing Small Read-Only Data Access

In the previous section, first the address of the memory was loaded before a second **LDR** instruction could load the actual data. This is necessary if the memory must be in SRAM; however, small bits of read-only memory with one **LDR** instruction can be loaded from the program section, typically flashed into the board’s ROM. This memory is only written to during the flash process but is fine to use for read-only data. For example:

```
          LDR R2, mynumber
          B   LOOP
mynumber: .WORD 0x1234ABCD
```

loads **R2** with the value 0x1234ABCD using only one **LDR** instruction. Notice that there is no equal sign before **mynumber** in the **LDR** instruction. This tells the Assembler to load the quantity directly and not create an indirection in the code section for it. The **mynumber** quantity must be defined in code and be reasonably close to the **LDR** instruction.

Generally, this is the fastest way to load registers with specific 32-bit numbers, and this is used extensively in Chapter 9.

Note Unless the program is relocated from ROM into RAM, it can’t be written back to this memory location when it runs.

As algorithms develop, an address is usually loaded once and used repeatedly, so most accesses take one instruction once going, such as indexing through memory in a loop.

Indexing Through Memory

All high-level programming languages have an array construct. They can define an array of objects and then access the individual elements by index. The high-level language will define the array with something like

```
DIM A[10] AS WORD
```

Then access the individual elements with statements like those in Listing 6-3.

Listing 6-3. Pseudocode to Loop Through an Array

```
// Set the 5th element of the array to the value 6
A[5] = 6
// Set the variable X equal to the 3rd array element
X = A[3]
// Loop through all 10 elements
FOR I = 1 TO 10
    // Set element I to I cubed
    A[I] = I ** 3
NEXT I
```

The ARM Cortex-M0 instruction set gives us support for doing these sorts of operations.

Suppose we have an array of ten words (4 bytes each) defined by

```
arr1: .FILL 10, 4, 0
```

Let's load the array's address into **R1**:

```
LDR R1, =arr1
```

We can now access the elements using **LDR** as demonstrated in Listing 6-4 and graphically represented in Figure 6-1.

Listing 6-4. Indexing into an Array

```
@ Load the first element
LDR R2, [R1]
@ Load element 3
@ The elements count from 0, so 2 is
@ the third one. Each word is 4 bytes,
@ so we need to multiply by 4
LDR R2, [R1, #(2 * 4)]
```

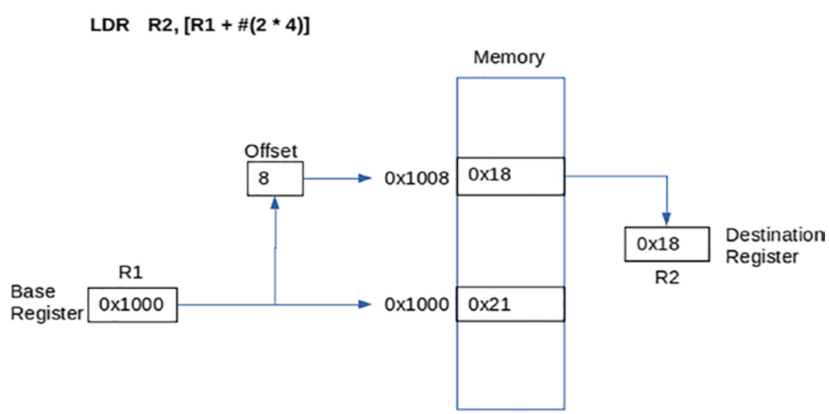

Figure 6-1. *Graphical view of using R1 and an index to load R2*

This is fine for accessing hard-coded elements, but what about via a variable? We can use a register as demonstrated in Listing 6-5.

Listing 6-5. Using a Register As an Offset

```
@ The 3rd element is still number 2
MOV R3, #(2 * 4)
@ Add the offset in R3 to R1 to get our element.
LDR R2, [R1, R3]
```

If we are incrementing through memory in a loop, we either increment the base address or increment the index register. Incrementing the base address is completed as follows:

```
LDR R2, [R1]     @ load the element R1 points to
ADD R1, #4       @ since each element is 4 bytes
```

Incrementing an index is similar:

```
LDR R2, [R1, R3]     @ load the element R1+R3 points to
ADD R3, #4           @ increment the index by the element size
```

The first method has the advantage that it uses one fewer register, and the second that we don't destroy the base memory address by incrementing it.

Note The immediate value with the **LDR** instruction is only 8 bits, so it can only be offset by 255 bytes. As a consequence, this is more often used to access structure elements as demonstrated in Chapter 9.

How to Store a Register

The Store Register **STR** instruction is a mirror of the **LDR** instruction. All the addressing modes discussed about for **LDR** work for **STR**. This is necessary since in a load-store architecture, everything loaded must be stored after it is processed in the CPU. The **STR** instruction was used a couple of times already in examples.

The **STR** instruction is simpler than the **LDR** instruction, since it isn't involved with building addresses. The **STR** instruction only saves using addresses that have already been constructed.

How to Convert to Uppercase

As an example of indexing through memory in loops, consider looping through a string of ASCII bytes. To convert any lowercase characters to uppercase, refer to Listing 6-6 that gives pseudocode to do this.

Listing 6-6. Pseudocode to Convert a String to Uppercase

```
i = 0
DO
      char = instr[i]
      IF char >= 'a' AND char <= 'z' THEN
            char = char - ('a' - 'A')
      END IF
      outstr[i] = char
      i = i + 1
UNTIL char == 0
PRINT outstr
```


In this example, we use NULL-terminated strings that are abundant in C programming. We used them for our **printf** strings as these are what the .asciz directive creates. The string is the sequence of characters, followed by a NULL (ASCII code 0 or \0) character. To process the string, we simply loop until we encounter the NULL character.

We've already covered **for** and **while** loops. The third common structured programming loop is the **DO/UNTIL** loop that puts the condition at the end of the loop. In this construct, the loop is always executed once. We want this, since if the string is empty, we still want to copy the NULL character, so the output string will then be empty as well. The algorithm in Listing 6-6 leaves the input string unchanged and produces a new output string with the uppercase version of the input string. As is common in Assembly Language processing, the logic is reversed to jump around the code in the IF block. Listing 6-7 shows the updated pseudocode.

Listing 6-7. Pseudocode on How We Will Implement the IF Statement

```
IF char < 'a' GOTO continue
IF char > 'z' GOTO continue
char = char - ('a' - 'A')
continue: // the rest of the program
```

We don't have the structured programming constructs of a high-level language to help us, and this turns out to be quite efficient in Assembly Language.

Listing 6-8 is the Assembly code to convert a string to uppercase.

Listing 6-8. Program to Convert a String to Uppercase

```
@
@ Assembler program to convert a string to
@ all upper case.
@
@ R0 - string parameter to printf
@ R3 - address of output string
@ R4 - address of input string
@ R5 - current character being processed
@

.thumb_func                @ Necessary because sdk uses BLX
.global main               @ Provide program starting address
                             to linker

main: BL stdio_init_all    @ initialize uart or usb

      LDR R4, =instr       @ start of input string
      LDR R3, =outstr      @ address of output string
@ The loop is until byte pointed to by R1 is non-zero
loop: LDRB R5, [R4]        @ load character
      ADD R4, #1           @ increment pointer
@ If R5 > 'z' then goto cont
      CMP R5, #'z'         @ is letter > 'z'?
      BGT cont
@ Else if R5 < 'a' then goto end if
      CMP R5, #'a'
      BLT cont             @ goto to end if
@ if we got here then the letter is lowercase, so convert it.
      SUB R5, #('a'-'A')
```



```
cont: @ end if
      STRB R5, [R3]        @ store character to output str
      ADD  R3, #1          @ increment pointer
      CMP  R5, #0          @ stop on hitting a null character
      BNE loop             @ loop if character isn't null

@ Setup the parameters to printf our upper case string
loop2: LDR R0, =outstr     @ string to print
       BL  printf          @ Call printf to output
       B   loop2
.data
instr:  .asciz  "This is our Test String that we will
convert.\n"
outstr: .fill   255, 1, 0
```

This program is quite short because besides all the comments and the code to print the string, there are only 13 Assembly Language instructions to initialize and execute the loop:

- **Two instructions**: Initialize our pointers for **instr** and **outstr**.
- **Five instructions**: Make up the **if** statement.
- **Six instructions**: For the loop, including loading character, saving a character, updating both pointers, checking for a null character, and branching if not null.

It would be nice if **STRB** also set the condition flags. **LDR** and **STR** just load and save. They don't have the functionality to examine what they are loading and saving, so they can't set the **CPSR**. Therefore, the need for the **CMP** instruction in the **UNTIL** part of the loop to test for NULL. In this example, we use the **LDRB** and **STRB** instructions since we are processing byte by byte.

To convert the letter to uppercase, we use

```
SUB R5, #('a'-'A')
```

The lowercase characters have higher values than the uppercase characters, so use an expression that the Assembler evaluates to get the correct number to subtract. Look at Listing 6-9, an abbreviated disassembly of our program.

Listing 6-9. Disassembly of the Uppercase Program

```
1000038a:       4c08            ldr   r4, [pc, #32]; (100003ac
                                      <cont+0x10>)
1000038c:       4b08            ldr   r3, [pc, #32]; (100003b0
                                      <cont+0x14>)

1000038e <loop>:
1000038e:       7825            ldrb  r5, [r4, #0]
10000390:       3401            adds  r4, #1
10000392:       2d7a            cmp   r5, #122; 0x7a
10000394:       dc02            bgt.n 1000039c <cont>
10000396:       2d61            cmp   r5, #97; 0x61
10000398:       db00            blt.n 1000039c <cont>
1000039a:       3d20            subs  r5, #32

1000039c <cont>:
1000039c:       701d            strb  r5, [r3, #0]
1000039e:       3301            adds  r3, #1
100003a0:       2d00            cmp   r5, #0
100003a2:       d1f4            bne.n 1000038e <loop>
100003a4:       4802            ldr   r0, [pc, #8]; (100003b0
                                      <cont+0x14>)
100003a6:       f003 fd0d       bl    10003dc4 <__wrap_printf>
```



```
100003ac:       2000025f        .word 0x2000025f; address of
                                instr
100003b0:       2000028e        .word 0x2000028e; address of
                                outstr

2000025f <instr>:
2000028e <outstr>:
```

The instruction

```
LDR R4, =instr
```

is converted to

```
ldr r4, [pc, #32] ;  (100003ac <cont+0x10>)
```

The comment tells us that **PC**+32 is the address 0x100003ac. We calculate that ourselves, if we take the address of the next instruction past this one (the one being decoded as this one executes), which is at 0x1000038c, and add 32 to get the same 0x100003ac.

This shows how the Assembler added the literal for the address of the string **instr** at the end of the code section. When we do the **LDR**, it accesses this literal and loads it into memory, and this gives us the address we need in memory. The other literal added to the code section is the address of **outstr**.

To see this program in action, it is worthwhile to single-step through it in **gdb**. You can watch the registers with the "i r" (info registers) command. To view **instr** and **oustr** as the processing occurs, there are a couple of ways of doing it. From the disassembly, we know the address of **instr** is 0x2000025f, so we can enter

```
(gdb) x /2s 0x2000025f
0x2000025f: "This is our Test String that we will convert.\n"
0x2000028e: "THI"
(gdb)
```

This is convenient since the **x** command knows how to format strings, but it doesn't know about labels. We can also enter

```
(gdb) p (char[10]) outstr
$8 = "TH\000\000\000\000\000\000\000"
(gdb)
```

The print (**p**) command knows about labels but doesn't know about data types. We must cast the label to tell it how to format the output. **Gdb** handles this better with high-level languages because it knows about the data types of the variables. In Assembly Language, we are closer to the metal. Next, we examine two instructions for loading and storing multiple registers at once.

How to Load and Store Multiple Registers

There are multiple register versions of all the **LDR** and **STR** instructions. The **LDM** and **STM** instructions take one register to use as the memory address and then a list of low registers (**R0–R7**) to load or store. The data needs to be contiguous, and the address register is updated to point after the data loaded or stored. For example, Listing 6-10 loads the address of a **dword** (the address is still 32 bits) and then loads the **dword** into **R2** and **R3**. Next, we store **R2** and **R3** back into **mydword2**.

Listing 6-10. Example of Loading and Storing Multiple Registers

```
        LDR R1, =mydword
        LDM R1!, {R2, R3}     @ load R2 & R3 from memory at R1
        STM R1!, {R2, R3}     @ store R2 & R3 to memory at R1
.data
mydword: .DWORD 0x1234567887654321
mydword2: .DWORD 0x0
```


The exclamation mark after the base register **R1!** indicates that this register will be updated as part of this operation—adding the length of the data to it. This is handy, since when used in a loop, you don't need an extra **ADD** instruction to update the memory address. In this case, **LDM** loads **mydword** into **R2** and **R3**, incrementing **R1** by 8 in the process. Next, the **STM** instruction writes **R2** and **R3** into memory location **mydword2**, again incrementing **R1** by 8.

Using this instruction, all the low registers **R0–R7** can be loaded or stored in one instruction. If one of the registers in the list is the base register, then it won't be incremented as part of the instruction. The Assembler gives a warning when this happens.

Summary

With this chapter completed, we can load data from memory, operate on it in the registers, and then save the result back to memory. We examined how the data load and store instructions to help with arrays of data and how they help us index through data in loops.

In the next chapter, we look at how to make code reusable. After all, wouldn't our uppercase program be handy if we could call it whenever needed?

Exercises

6-1. Create a small program to try out all the data definition directives the Assembler provides. Assemble your program and examine the data in the disassembly listing. Add some align directives and examine how they move around.

6-2. Explain how the **LDR** instruction lets any 32-bit address load in only one 16-bit instruction.

6-3. Write a program that converts a string to all lowercase.

6-4. Write a program that converts any nonalphabetic character in a NULL-terminated string to a space.

CHAPTER 7

How to Call Functions and Use the Stack

In this chapter, we examine how to organize code into small independent units called **functions**. This allows us to build reusable components that we can call easily from anywhere we wish. Typically, in software development, we start with low-level components and then build on these to create higher-level applications. So far, we learned how to loop, perform conditional logic, and perform some arithmetic. Now, we examine how to compartmentalize our code into building blocks.

We introduce the **stack**; this is a computer science data structure for storing data. If we are going to build useful reusable functions, we will need a good way to manage register usage so that all these functions don't clobber each other. In Chapter 5, we studied how to store data in main memory. The problem with this is that this memory exists for the duration that our program runs. With small functions, like converting to an uppercase program, they run quickly and might need a few memory locations while they run, but when they are done, they don't need this memory anymore. Stacks provide us a tool to manage register usage across function calls and a tool to provide memory to functions for the duration of their invocation.

We introduce several low-level concepts first, and then we put them all together to effectively create and use functions. We start with stacks and their support on the RP2040.

S. Smith, *RP2040 Assembly Language Programming*,
https://doi.org/10.1007/978-1-4842-7753-9_7

About Stacks on the RP2040

In computer science, a stack is an area of memory where there are two operations:

- **push**: Adds an element to the area
- **pop**: Returns and removes the element that was most recently added

This behavior is also called a **LIFO** (last in first out) queue.

When a program runs from the RP2040, the size of the stack is configurable, by default 0x800 (2048 words). In Chapter 2, we mentioned that register **R13** had a special purpose as the stack pointer (**SP**). You might have noticed that **R13** is named **SP** in **gdb**, and you might have noticed that when you debugged programs, it had a large value, like 0x20041fe0. This is a pointer to the current stack location.

There are two instructions to save register values to the stack and then restore those values. These are

PUSH {reglist}

POP {reglist}

The {reglist} parameter is a list of registers, containing a comma-separated list of registers and register ranges. A register range is something like **R2–R4**, which means **R2**, **R3**, and **R4**, for example:

```
PUSH {r0, r5-r7, LR}
POP {r0-r4, r6, PC}
```

The registers are stored on the stack in numerical order, with the lowest register at the lowest address. You can include any low register (**R0–R7**) as well as **LR** in the **PUSH** instruction and **PC** in the **POP** instruction. We'll see why this functionality for **LR** and **PC** is useful shortly. Figure 7-1 shows the process of pushing a register onto the stack, and Figure 7-2 shows the reverse operation of popping that value off the stack.

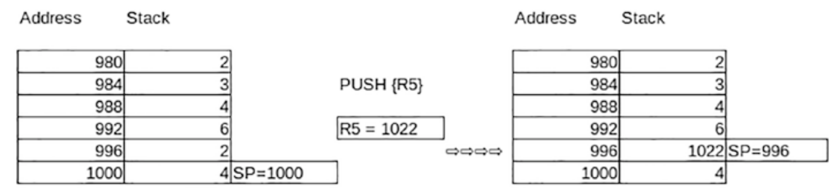

Figure 7-1. *Pushing R5 onto the stack*

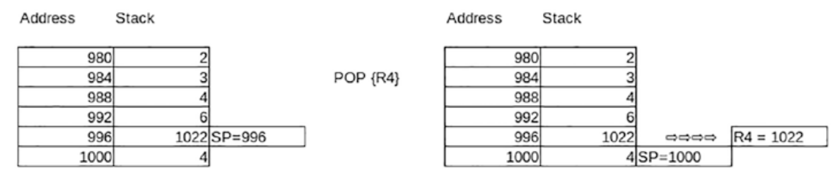

Figure 7-2. *Popping R4 from the stack*

Before we make use of these instructions, we need to call and return from functions.

How to Branch with Link

To call a function, first set up the ability for the function to return execution to after the point where the function is called. This is done with the other special register listed in Chapter 2, the Link Register (**LR**), which is **R14**. To make use of **LR**, enter the Branch with Link (**BL**) instruction, which is the same as the Branch (**B**) instruction, except it puts the address of the next instruction into **LR** before it performs the branch, giving a mechanism to return from the function.

One way to return from a function is to use the Branch and Exchange (**BX**) instruction. This branch instruction takes a register as its argument, allowing it to branch to the address stored in **LR** to continue processing after the function completes.

In Listing 7-1, the **BL** instruction stores the address of the following **MOV** instruction into **LR** and then branches to myfunc. Myfunc does the useful work the function was written to do and then returns execution to the caller by having **BX** branch to the location stored in **LR**, which is the **MOV** instruction following the **BL** instruction.

Listing 7-1. Skeleton Code to Call a Function and Return

```
     @ ... other code ...
     BL  myfunc
     MOV R1, #4
     @ ... more code ...
-----------------------------
myfunc: @ do some work
     BX  LR
```

This works for functions that are one level deep, but what if the function needs to call other functions?

About Nesting Function Calls

We successfully called and returned from a function, but we never used the stack. Why did we introduce the stack first and then not use it? First, think what happens if during its processing myfunc calls another function. This is fairly common, as we write code building on the functionality previously written. If myfunc executes a **BL** instruction, then **BL** copies the next address into **LR** overwriting the return address for myfunc, and myfunc won't be able to return. What we need is a way to keep a chain of return addresses as we call function after function. Rather not a chain of return addresses, but a stack of return addresses.

If myfunc is going to call other functions, then it needs to push **LR** onto the stack as the first thing it does and pop it from the stack just before it returns. However, there is a problem here, because you can push **LR**,

but you can't **POP** it. Instead, you can **POP** the **PC**. The reason is that this saves you an instruction on returning from functions. **POP PC** loads the saved value of **LR** directly into the **PC**, causing the processor to jump to that memory location. Listing 7-2 shows this process.

Listing 7-2. Skeleton Code for a Function That Calls Another Function

```
     @ ... other code ...
     BL  myfunc
     MOV R1, #4
     @ ... more code ...
-----------------------------
myfunc: PUSH {LR}
     @   do some work ...
     BL  myfunc2
     @   do some more work...
     POP {PC}
myfunc2: @ do some work ....
     BX  LR
```

In this example, we see how convenient it is to store data to the stack that only needs to exist for the duration of a function call.

If a function, such as myfunc, calls other functions, then it must save **LR**; however, if it doesn't call other functions, such as myfunc2**,** then it doesn't need to save **LR**. Programmers often push **LR** regardless, since if the function is modified later to add a function call and the programmer forgets to add **LR** to the list of saved registers, then the program fails to return and either goes into an infinite loop or crashes. The downside is that there is only so much bandwidth between the CPU and memory, so to **PUSH** and **POP** more registers does take extra execution cycles. The trade-off in speed vs. maintainability is a subjective decision depending on the circumstances.

When you work in high-level programming languages, you know that functions take parameters and return results and the same is true in Assembly Language.

About Function Parameters and Return Values

In high-level languages, functions take parameters and return their results, and Assembly Language programming is no different. We could invent our own mechanisms to do this, but this is counterproductive. Eventually, we want our code to interoperate with code written in other programming languages. We will want to call new, superfast functions from C code and might want to call functions written in C, such as those in the RP2040 SDK.

To facilitate this, there are a set of design patterns for calling functions. If we follow these patterns, our code will work reliably since others have already worked out all the bugs, plus we achieve the goal of writing interoperable code.

The caller passes the first four parameters in **R0**, **R1**, **R2**, and **R3**. If there are additional parameters, then they are pushed onto the stack. If we only have two parameters, then we would only use **R0** and **R1**. This means the first four parameters are already loaded into registers and ready to be processed. Additional parameters need to be popped from the stack before being processed.

To return a value to the caller, place it in **R0** before returning. If you need to return more data, you will have one of the parameters be an address to a memory location where you can place the additional data to be returned. This is the same as C where you return data through call by reference parameters.

The RP2040 only contains 16 registers, and most instructions only work with eight of these. How do we ensure that our registers aren't wiped out when we call a function? This is the topic of the next section.

How to Manage the Registers

If you call a function, chances are it was written by a different programmer, and you don't know what registers it will use. It would be very inefficient if you had to reload all your registers every time you call a function. As a result, there are a set of rules to govern which registers a function can use and who is responsible for saving each one:

- **R0–R3**: These are the function parameters. The function can use these for any other purpose modifying them freely. If the calling routine needs them saved, it must save them itself.
- **R4–R11**: These can be used freely by the called routine, but if it is responsible for saving them. That means the calling routine can assume these registers are intact.
- **R12**: This is the intraprocedure call scratch register and shouldn't be used. If you do, some SDK functionality (like **printf**) will not work until you restore it.
- **SP**: This can be freely used by the called routine. The routine must **POP** the stack the same number of times that it **PUSH**es, so it is intact for the calling routine.
- **LR**: The called routine must preserve this as we discussed in the last section.
- **CPSR**: Neither routine can make any assumptions about the **CPSR**. As far as the called routine is concerned, all the flags are unknown; similarly, they are unknown to the caller when the function returns.

With all this, we can now summarize the function call algorithm.

Summary of the Function Call Algorithm

Calling routine

1. If we need any of **R0–R4**, save them.
2. Move the first four parameters into registers **R0–R4**.
3. Push any additional parameters onto the stack.
4. Use **BL** to call the function.
5. Evaluate the return code in **R0**.
6. Restore any of **R0–R4** that we saved.

Called function

1. **PUSH LR** and **R4–R11** onto the stack.
2. Do our work.
3. Put our return code into **R0**.
4. **POP PC** and **R4–R11**.

Note Saving all of **LR** and **R4–R11** is the safest and most maintainable practice. However, if we don't use some of these registers, skip saving them to save some execution time on function entry and exit. Further, the **PUSH** and **POP** instructions do not work with high registers **R8–R11**; therefore, to save these on the stack, move them to low registers and then use **PUSH** and **POP**. This is one reason why the high registers are rarely used.

To save some steps, just use **R0–R3** for function parameters and return codes and short-term work; then the calling routine never has to save and restore them around function calls.

We've assumed all parameters are 32 bits here. The rule is that if something is less than 32 bits, place it in a 32-bit register or stack location to pass it. If the parameter is larger than 32 bits, break it up into multiple 32-bit chunks and treat it as multiple parameters. For larger items, passing by reference is usually easier (passing an address to the parameter).

Now that we've been introduced to all the branch instructions, let's summarize and note some extra, perhaps unexpected, functionality.

More on the Branch Instructions

These are the branch instructions supported by the ARM Cortex-M0+ CPU:

1. B label
2. B{condition} label
3. BX Rm
4. BL label
5. BLX Rm

- Numbers 1 and 2 are 16-bit instructions, and the label is an offset from the **PC**. Their range is -2048 to 2046 from the current program location. This makes them appropriate for loops and jumps within single functions. This prevents writing large single routines that jump madly about.

- Number 4 is one of the six 32-bit instructions supported by the ARM Cortex-M0+. This is a **PC** relative offset, but the range is -16777216 to 16777214, which is larger than the amount of memory contained in either SRAM or Flash on all current RP2040 boards. This means you can reliably call any routine in your program or the SDK without issue.
- Numbers 3 and 5 are the two forms that jump indirectly to an address contained in register **Rm**. This register can be any high or low register except the **PC**. Since the address is formed in a register, it can be anywhere within the RP2040's full 32-bit address space.

There is a bit more complexity around the **BX** and **BLX** instructions that we cover next.

About the X Factor

If you look in ARM's Cortex-M0+, the **BX** instruction is called the Branch and Exchange instruction, which makes you question what we are exchanging. In the full ARM A-series processors, like those used in the Raspberry Pi 4, when running in 32-bit mode, there are two separate sets of instructions:

1. The regular 32-bit length instructions
2. The 16-bit "thumb" instructions, which include a small number of 32-bit instructions

The exchange in the **BX** and **BLX** instructions is the mechanism to switch between these two instruction sets. This allows code of type 1 to call code of type 2 and vice versa. The RP2040 only supports type 2 instructions, but there is only one instruction set, so why are we discussing

this? The problem to be careful of is that if we indicate to **BX** or **BLX** that we want to switch instruction sets to type 1, then the RP2040 throws a hardware fault, and the program terminates.

Since all instructions have to be aligned on either 32-bit or 16-bit boundaries, the address of all instructions is even. This means the low-order bit in the register containing the memory address to jump to is unused. To keep things compact, the ARM processor uses every bit possible, so it uses this bit to indicate the instruction set type. If the low-order bit is even, then it switches to type 1, full 32-bit instruction mode, and if the address is odd, then it switches to type 2, 16-bit thumb mode. The problem is that addresses are usually even and if we don't do anything, then the Assembler generates even addresses and the RP2040 generates a hardware fault when it tries to jump. This is why we have to put

```
.thumb_func
```

before our definition of the function main.

The SDK calls main with a **BLX** instruction, and .thumb_func tells the Assembler to set the low-order bit to one for this address. We do the same thing for any address that we call with either **BX** or **BLX**.

In the uppercase function that we study next, we will see that the **BL** instruction sets the low-order bit in the return address it places in **LR** so that it returns correctly when **BX** is used.

Uppercase Revisited

Let's organize our uppercase example from Chapter 6 as a proper function. We'll move the function into its own file and modify the CMakeLists.txt to make both the calling program and the uppercase function.

First, create a file called main.S containing Listing 7-3 for the driving application.

Listing 7-3. Main Program for Uppercase Example

```
@
@ Assembly Language program to convert a string to
@ all upper case by calling a function.
@
@ R0 - parameters to printf
@ R1 - address of output string
@ R0 - address of input string
@ R5 - current character being processed
@

.thumb_func                  @ Necessary because sdk uses BLX
.global main                 @ Provide program starting address

main: BL stdio_init_all      @ initialize uart or usb

repeat:
      LDR R0, =instr         @ start of input string
      LDR R1, =outstr        @ address of output string
      MOV R4, #12
      MOV R5, #13

      BL  toupper

      LDR R0, =outstr        @ string to print
      BL  printf

      B   repeat             @ loop forever

.data
instr:  .asciz  "This is our Test String that we will
convert.\n"
outstr: .fill   255, 1, 0
```

Now create a file called upper.S containing Listing 7-4, the uppercase conversion function.

Listing 7-4. Function to Convert Strings to All Uppercase

```
@
@ Assembly Langauge function to convert a string to
@ all upper case.
@
@ R1 - address of output string
@ R0 - address of input string
@ R4 - original output string for length calc.
@ R5 - current character being processed
@

.global toupper              @ Allow other files to call this
                               routine

toupper: PUSH {R4-R5}        @ Save the registers we use.
         MOV R4, R1
@ The loop is until byte pointed to by R1 is non-zero
loop: LDRB R5, [R0]          @ load character
      ADD  R0, #1            @ increment instr pointer
@ If R5 > 'z' then goto cont
      CMP  R5, #'z'          @ is letter > 'z'?
      BGT cont
@ Else if R5 < 'a' then goto end if
      CMP  R5, #'a'
      BLT  cont              @ goto to end if
@ if we got here then the letter is lowercase, so convert it.
      SUB R5, #('a'-'A')
cont: @ end if
      STRB R5, [R1]          @ store character to output str
      ADD  R1, #1            @ increment outstr pointer
      CMP  R5, #0            @ stop on hitting a null character
      BNE  loop              @ loop if character isn't null
```

```
    SUB  R0, R1, R4        @ get the length by subtracting
                             the pointers
    POP {R4-R5}            @ Restore the register we use.
    BX   LR                @ Return to caller
```

To build these, use the CMakeLists.txt file in *Listing 7-5*.

Listing 7-5. Makefile for the uppercase function example

```
cmake_minimum_required(VERSION 3.13)

include(pico_sdk_import.cmake)
project(Functions C CXX ASM)

set(CMAKE_C_STANDARD 11)
set(CMAKE_CXX_STANDARD 17)

pico_sdk_init()

include_directories(${CMAKE_SOURCE_DIR})

add_executable(Functions
  main.S
  upper.S
 )

pico_enable_stdio_uart(Functions 1)
pico_enable_stdio_usb(Functions 0)

pico_add_extra_outputs(Functions)

target_link_libraries(Functions pico_stdlib)
```

Let's step through the function call to examine the contents of important registers and the stack. We set a breakpoint at main and single-step through the first couple of instructions and stop at the **BL** instruction. I set **R4** to 12 and **R5** to 13, so we can follow how these are saved to the stack.


```
R4 0xc 12
R5 0xd 13
Sp 0x20042000 0x20042000
Lr 0x10003f67 268451687
Pc 0x10000368 0x10000368 <repeat+8>
```

We see the **BL** instruction is at 0x10000368. Now let's single-step again to execute the **BL** instruction. Here are the same registers:

```
R4 0xc 12
R5 0xd 13
Sp 0x20042000 0x20042000
Lr 0x1000036d 268436333
Pc 0x100003d2 0x100003d2 <toupper>
```

The **LR** has been set to 0x1000036d, which is the instruction after the **BL** instruction (0x10000368+5); this is 4 bytes for the length of the **BL** instruction plus 1 more to indicate we are returning to 16-bit instructions. The **PC** is now 0x100003d2, pointing to the first instruction in the toupper routine. The first instruction in toupper is the **PUSH** instruction to save registers **R4** and **R5**. Let's single-step through that instruction and examine the registers again.

```
R4 0xc 12
R5 0xd 13
Sp 0x20041ff8 0x20041ff8
Lr 0x10088 65672
Pc 0x100003d4 0x100003d4 <toupper+2>
```

We see that the stack pointer (**SP**) has been decremented by 8 bytes (two words) to 0x20041ff8. None of the other registers have changed. Pushing registers onto the stack does not affect their values; it only saves them. If we look at location 0x20041ff8, we see

```
(gdb) x /4xw 0x20041ff8
0x20041ff8: 0x0000000c 0x0000000d 0x00000000 0x00000000
(gdb)
```

We see copies of registers **R4** and **R5** on the stack and that **SP** points to the last item saved (and not the next free slot).

Note The toupper function doesn't call any other functions, so we don't save **LR** along with **R4** and **R5**. If we ever change it to do so, we will need to add **LR** to the list. This version of toupper is intended to be as fast as possible, so I didn't add any extra code for future maintainability and safety.

Most C programmers will object that this function is dangerous. If the input string isn't NULL terminated, then it will overrun the output string buffer, overwriting the memory past the end. The solution is to pass in a third parameter with the buffer lengths and check in the loop that we stop at the end of the buffer if there is no NULL character.

This routine only processes the core ASCII characters. It doesn't handle the localized characters like é; it won't be converted to É.

This was a simple routine; most functions have several internal variables that require storage, often more than fit in the registers, leading to the need for stack frames.

About Stack Frames

In our uppercase function, we didn't need any additional memory, since we could do all our work with the available registers. When we code larger functions, we often require more memory for our variables than fit in the registers. Rather than add clutter to the **.data** section, we store these variables on the stack.

PUSHing these variables on the stack isn't practical, since we usually need to access them in a random order, rather than the strict **LIFO** protocol that **PUSH**/**POP** enforces.

To allocate space on the stack, use a subtract instruction to grow the stack by the amount needed. Suppose we need three variables that are each 32-bit integers, say, a, b, and c. Therefore, we need 12 bytes allocated on the stack (3 variables × 4 bytes/word).

```
SUB SP, #12
```

This moves the stack pointer down by 12 bytes, providing us a region of memory on the stack to place our variables. Suppose a is in **R0**, b in **R1**, and c in **R2**; we can then store these using the following:

```
STR R0, [SP]     @ Store a
STR R1, [SP, #4] @ Store b
STR R2, [SP, #8] @ Store c
```

Before the end of the function, we need to execute the following:

```
ADD SP, #12
```

To release our variables from the stack. Remember, it is the responsibility of a function to restore **SP** to its original state before returning. Next, let's look at an example.

Stack Frame Example

Listing 7-6 is a simple skeletal example of a function that creates three variables on the stack and shows how to use them. It isn't intended to be a working program, just demonstrating how to define and access variables.

Listing 7-6. Simple Skeletal Function That Demonstrates a Stack Frame

```
@ Simple function that takes 2 parameters
@ VAR1 and VAR2. The function adds them,
@ storing the result in a variable SUM.
@ The function returns the sum.
@ It is assumed this function does other work,
@ including other functions.
@ Define our variables
     .EQU VAR1, 0
     .EQU VAR2, 4
     .EQU SUM, 8
SUMFN: PUSH {R4-R7, LR}
     SUB SP, #12             @ room for three 32-bit values
     STR R0, [SP, #VAR1]     @ save passed in param.
     STR R1, [SP, #VAR2]     @ save second param.
@ Do a bunch of other work, but don't change SP.
     LDR R4, [SP, #VAR1]
     LDR R5, [SP, #VAR2]
     ADD R6, R4, R5
     STR R6, [SP, #SUM]
@ Do other work
@ Function Epilog
     LDR R0, [SP, #SUM]     @ load sum to return
     ADD SP, #12            @ Release local vars
     POP {R4-R7, PC}        @ Restore regs and return
```


We introduced a new concept in this example—symbols via the **.EQU** directive.

How to Define Symbols

In this example, we introduce the **.EQU** Assembler directive. This directive allows us to define symbols that will be substituted by the Assembler before generating the compiled code. This way, we can make the code more readable. In this example, keeping track of which variable is which on the stack makes the code hard to read and is error prone. With the .EQU directive, we can define each variable's offset on the stack once.

Sadly, .EQU only defines numbers, so we can't define the whole "[SP, #4]" type string.

Functions aren't the only way to make reusable code; next, we look at macros.

How to Create Macros

Another way to make our uppercase loop into a reusable bit of code is to use macros. The GNU Assembler has a powerful macro capability with macros rather than calling a function. The Assembler creates a copy of the code in each place where it is called, substituting any parameters. Consider this alternate implementation of our uppercase program, where the first file is mainmacro.S containing the contents of Listing 7-7.

Listing 7-7. Program to Call Our Toupper Macro

```
@
@ Assembler program to convert a string to
@ all upper case by calling a function.
@
@ R0 - parameters to printf
```

```
@ R1 - address of output string
@ R0 - address of input string
@

.include "uppermacro.S"

.global mainmacro     @ Provide function starting address

mainmacro: PUSH {LR}

      toupper tststr, buffer

      LDR R0, =buffer @ string to print
      BL  printf

      toupper tststr2, buffer

      LDR R0, =buffer @ string to print
      BL  printf

      POP {PC}

.datakhconvert.\n"
tststr2: .asciz   "A second string to uppercase!!\n"
buffer:  .fill    255, 1, 0
```

Since we know how to set things up as functions, we set up the mainmacro.S code as a function and call it from main.S with

```
@ Call macro version.
BL mainmacro
```

This way we only need one project for this chapter's sample code. These new files are also added to CMakeLists.txt.

The macro to uppercase the string is in uppermacro.S containing Listing 7-8.

Listing 7-8. Macro Version of the Toupper Function

```
@
@ Assembler program to convert a string to
@ all uppercase (implemented as a macro)
@
@ R1 - address of output string
@ R0 - address of input string
@ R2 - original output string for length calc.
@ R3 - current character being processed
@

@ label 1 = loop
@ label 2 = cont

.MACRO toupper instr, outstr
       LDR R0, =\instr
       LDR R1, =\outstr
       MOV R2, R1
@ The loop is until byte pointed to by R1 is non-zero
1:    LDRB R3, [R0]    @ load character
      ADD  R0, #1      @ increment instr poitner
@ If R5 > 'z' then goto cont
      CMP R3, #'z'     @ is letter > 'z'?
      BGT 2f
@ Else if R5 < 'a' then goto end if
      CMP R3, #'a'
      BLT 2f  @ goto to end if
@ if we got here then the letter is lowercase, so convert it.
      SUB R3, #('a'-'A')
2:    @ end if
      STRB R3, [R1]    @ store character to output str
      ADD  R1, #1      @ increment outstr pointer
```

```
    CMP  R3, #0      @ stop on hitting a null character
    BNE  1b          @ loop if character isn't null
    SUB  R0, R1, R2  @ get the length by subtracting the
                       pointers
.ENDM
```

The first new concept is the **.include** directive.

About Include Directive

The file uppermacro.S defines our macro to convert a string to uppercase. The macro doesn't generate any code; it just defines the macro for the Assembler to insert wherever it is called from. This file doesn't generate an object (*.o) file; rather, it is included by whichever file needs to use it.

The **.include** directive

```
.include "uppermacro.S"
```

takes the contents of this file and inserts it at this point so that our source file becomes larger. This is done before any other processing. This is similar to the C #include preprocessor directive.

Now that we know how to include our macro, let's look at how to define macros.

How to Define a Macro

A macro is defined with the .**MACRO** directive. This gives the name of the macro and lists its parameters. The macro ends at the following **.ENDM** directive. The form of the directive is

```
.MACRO macroname parameter1, parameter2, ...
```


Within the macro, you specify the parameters by preceding their name with a backslash, for instance, **parameter1** to place the value of parameter1. The toupper macro defines two parameters, **instr** and **outstr**:

```
.MACRO toupper instr, outstr
```

You can see how the parameters are used in the code with \instr and \oustr. These are text substitutions and need to result in correct Assembly Language syntax or you will get an error.

In the code, the labels are replaced by numbers—why is that?

About Labels

The labels "loop" and "cont" are replaced with the labels "1" and "2". This takes away from the readability of the program. The reason to do this is that if we didn't, we get an error that a label is defined more than once if the macro is used more than once. The strategy here is that the Assembler lets numeric labels be defined as many times as you want. To reference them in our code, we used

```
BGT 2f
BNE 1b @ loop if character isn't null
```

The **f** after the **2** means the next label **2** is in the forward direction. The **1b** means the next label **1** is in the backward direction.

To prove that this works, we call toupper twice in the mainmacro.S file to show that everything works and that this macro can be reused as many times as we like. But why would we want to use macros over functions?

Why Macros?

Macros substitute a copy of the code at every point they're used. This makes an executable file larger. Look at the disassembly file for this project, and you will see the two copies of code inserted. With functions, there is no extra code generated each time. This is why functions are appealing, even with the extra work of dealing with the stack.

The reason macros get used is performance. The RP2040 runs at 133MHz, which isn't that fast by modern standards. Remember that whenever we branch, we have to restart the execution pipeline, making branching an expensive instruction. With macros, we eliminate the **BL** branch to call the function and the **BX** branch to return. We also eliminate the **PUSH** and **POP** instructions to save and restore any registers used. If a macro is small and used a lot, there could be considerable execution time savings.

Note Notice in the macro implementation of toupper that only registers **R0–R3** are used. This is to avoid using any registers important to the caller. There is no standard on how to regulate register usage with macros, like there is with functions, so it is up to the programmer to avoid conflicts and strange bugs.

Summary

In this chapter, we covered the ARM stack and how it is used to help implement functions. We covered how to write and call functions as a first step to creating libraries of reusable code. We learned how to manage register usage, so there aren't any conflicts between calling programs and functions. We learned the function calling protocol that allows us to interoperate with other programming languages. We looked at defining stack-based storage for local variables and how to use this memory.

Finally, we covered the GNU Assembler's macro ability as an alternative to functions in certain performance critical applications.

Next, in Chapter 8, is more detail at calling and being called by C routines, in particular, how to interact with the RP2040's SDK.

Exercises

7-1. Suppose we have a function that uses registers **R4**, **R5**, **R6**, **R8**, and **R9**. Further, this function calls other functions. Code the prologue and epilogue of this function to store and restore the correct registers to/from the stack. Be careful how you handle the high registers **R8** and **R9**.

7-2. Write a function to convert text to all lowercase. Have this function in one file and a main program in another file. In the main program, call the function three times with different test strings.

7-3. Convert the lowercase program in Exercise 7-2 to a macro. Have it run on the same three test strings to ensure it works properly.

7-4. Why does the function calling protocol have some registers need to be saved by the caller and some by the callee? Why not make all saved by one or the other?

7-5. Why would the SDK call the main routine with a **BLX** instruction rather than a **BL** instruction?

CHAPTER 8

Interacting with C and the SDK

In the early days of microcomputers, like the Apple II, people wrote complete applications in Assembly Language, such as the first spreadsheet program VisiCalc. Many video games were written in Assembly Language to squeeze every bit of performance they could out of the hardware. Modern compilers, like the GNU C compiler, generate adequate code, and microcontrollers, like the RP2040, are much faster. As a result, most applications are written in a collection of programming languages, where each excels at a specific function.

The RP2040 SDK contains a wealth of efficient code, and we want to use that as much as possible rather than writing everything from scratch. Most of the SDK is written in C, but there are quite a few Assembly Language routines that we can study.

In this chapter, we look at using components written in C/C++ from our Assembly Language code and at how other languages can make use of the fast-efficient code we are writing in Assembly Language.

With this chapter, we use the Raspberry Pi Pico's hardware I/O capabilities. We describe how to set up three flashing LEDs and then control them using different techniques over the following two chapters. In this chapter, we control the LEDs using the RP2040's SDK. This gives us more experience using C functions and the extra complexity present in the SDK.

S. Smith, *RP2040 Assembly Language Programming*,
https://doi.org/10.1007/978-1-4842-7753-9_8

How to Wire Flashing LEDs

Before writing programs, we need to wire the circuitry to connect LEDs to a breadboard. For this project we require

- Three 220Ω resistors (red, red, black)
- Three LEDs (preferably of different colors)
- Four connecting wires

This assumes you've soldered pins to your RP2040 board and plugged it into a breadboard as outlined in Chapter 1. These parts are typically included in any Raspberry Pi or Arduino electronics starter kit.

We will connect each of three LEDs to a GPIO pin, in this case, 18, 19, and 20, and then to ground through a resistor. We need the resistor because the GPIO is specified to keep the current under 16mA, or the circuits can be damaged. Most of the kits come with several 220 ohm resistors. By Ohm's law, I = V / R, these would cause the current to be 3.3V/220Ω = 15mA, so just right. The resistor needs to be in series with the LED, since the LED's resistance is quite low (typically around 13 ohms and variable).

Warning LEDs have a positive and negative side. The positive side must connect to the GPIO pin; reversing it could damage the LED.

Figure 8-1 shows how the LEDs and resistors are wired on a breadboard.

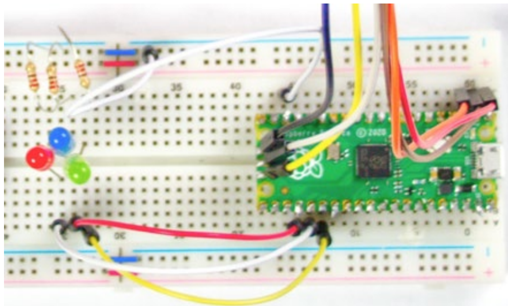

Figure 8-1. *Breadboard with LEDs and resistors installed*

With the hardware wired, it's the time to write some code.

How to Flash LEDs with the SDK

In this chapter, we flash the LEDs using functions in the RP2040's SDK. In later chapters, we repeat this process using Assembly Language to write to the hardware directly and then using the RP2040's PIO coprocessors to offload the work from the CPU. Using the SDK is easiest, since well-tested functions do the work for us. This is a typical process in writing code for microprocessors; first, write the program the easiest way, then identify parts that aren't performant and rewrite those in Assembly Language, or use coprocessors to create a better experience.

In this example, we use four SDK functions:

1. **void gpio_init (uint gpio)**: Initialize a pin for GPIO. Many pins have multiple functions.
2. **static void gpio_set_dir (uint gpio, bool out)**: Set the direction of the pin, either input or output.
3. **static void gpio_put (uint gpio, bool value)**: Set a GPIO pin either high or low.
4. **void sleep_ms (uint32_t ms)**: Sleep for the specified number of milliseconds.

C functions follow the calling convention that we learned in Chapter 7; therefore, we know to place the first parameter in **R0** and the second parameter in **R1**. None of these functions return a value, so we don't need to check **R0** after making the call. Basically, we do the following:

1. Initialize the three GPIO pins: 18, 19, and 20.
2. Sequentially turn on a LED.
3. Sleep for 1/5th of a second.
4. Turn off the LED.

Listing 8-1 contains the Assembly Language source code for this, which should be placed in the file flashledssdk.S.

Listing 8-1. Assembly Language Source Code to Flash the LEDs Using the SDK

```
@
@ Assembler program to flash three LEDs connected to
@ the Raspberry Pi Pico GPIO port using the Pico SDK.
@

     .EQU LED_PIN1, 18
     .EQU LED_PIN2, 19
     .EQU LED_PIN3, 20
     .EQU GPIO_OUT, 1
     .EQU sleep_time, 200

.thumb_func      @ Necessary because sdk uses BLX
.global main     @ Provide program starting address

main:
     MOV R0, #LED_PIN1
     BL  gpio_init
     MOV R0, #LED_PIN1
```

```
      MOV R1, #GPIO_OUT
      BL  link_gpio_set_dir
      MOV R0, #LED_PIN2
      BL  gpio_init
      MOV R0, #LED_PIN2
      MOV R1, #GPIO_OUT
      BL  link_gpio_set_dir
      MOV R0, #LED_PIN3
      BL  gpio_init
      MOV R0, #LED_PIN3
      MOV R1, #GPIO_OUT
      BL  link_gpio_set_dir
loop: MOV R0, #LED_PIN1
      MOV R1, #1
      BL  link_gpio_put
      LDR R0, =sleep_time
      BL  sleep_ms
      MOV R0, #LED_PIN1
      MOV R1, #0
      BL  link_gpio_put
      MOV R0, #LED_PIN2
      MOV R1, #1
      BL  link_gpio_put
      LDR R0, =sleep_time
      BL  sleep_ms
      MOV R0, #LED_PIN2
      MOV R1, #0
      BL  link_gpio_put
      MOV R0, #LED_PIN3
      MOV R1, #1
      BL  link_gpio_put
      LDR R0, =sleep_time
```



```
      BL  sleep_ms
      MOV R0, #LED_PIN3
      MOV R1, #0
      BL  link_gpio_put
      B   loop
```

In this program, we call link_gpio_put and link_gpio_set_dir rather than gpio_put and gpio_set_dir directly. Look in the SDK to find gpio_put defined in gpio.h as

```
static inline void gpio_set_dir(uint gpio, bool out) {
    uint32_t mask = 1ul << gpio;
    if (out)
        gpio_set_dir_out_masked(mask);
    else
        gpio_set_dir_in_masked(mask);
}
```

The problem is that this function is defined as **inline**. This tells the C compiler that this isn't a function and to insert the code inline wherever it is called. This is the same as what we did with macros in Chapter 7. Since this isn't a function, just a snippet of C code, it can't be called directly from the Assembly Language code because there is nothing to call. This leads to Listing 8-2, where a C file can be provided that wraps this inline C code and exposes them as functions that can be called.

Listing 8-2. C Wrapper Functions for the Inline Code We Need from the SDK

```
/* C wrapper functions for the RP2040 SDK
 * Incline functions gpio_set_dir and gpio_put.
 */

#include "hardware/gpio.h"
```

```
void link_gpio_set_dir(int pin, int dir)
{
     gpio_set_dir(pin, dir);
}

void link_gpio_put(int pin, int value)
{
     gpio_put(pin, value);
}
```

Note This is preferable to editing the source code in the SDK to remove the inline keyword, as it would cause problems getting newer versions of the SDK.

The CMakeLists.txt file is given in Listing 8-3 and is standard.

Listing 8-3. CMakeLists.txt File for This Project

```
cmake_minimum_required(VERSION 3.13)

include(pico_sdk_import.cmake)
project(test_project C CXX ASM)

set(CMAKE_C_STANDARD 11)
set(CMAKE_CXX_STANDARD 17)

pico_sdk_init()

include_directories(${CMAKE_SOURCE_DIR})

add_executable(FlashLEDsSDK
  flashledssdk.S
  sdklink.c
)

pico_enable_stdio_uart(FlashLEDsSDK 1)
```



```
pico_add_extra_outputs(FlashLEDsSDK)

target_link_libraries(FlashLEDsSDK pico_stdlib)
```

With these files, follow the procedures in Chapter 1 to build the **uf2** file and copy it to your Raspberry Pi Pico. The LEDs should flash in turn quickly over and over again. If the program doesn't work, then create a debug build and step through the program in **gdb**.

We'll learn new approaches to functions like **gpio_put** in the following chapters, but initialization functions like **gpio_init** are typically not time critical and you may as well make use of the SDK, rather than creating your own.

How to Call Assembly Routines from C

A typical scenario is to write most of the application in C and then call Assembly Language routines in specific use cases. If we follow the function calling protocol from Chapter 7, C won't be able to tell the difference between our functions and any other functions written in C.

As an example, let's call the toupper function from Chapter 7 and call it from C. Listing 8-4 contains the C code for **uppertst.c** to call our Assembly Language function.

Listing 8-4. Main Program to Show Calling Our Toupper Function from C

```
//
// C program to call our Assembly Language
// toupper routine.
//

#include <stdio.h>
#include "pico/stdlib.h"
```

```
extern int mytoupper( char *, char * );

#define MAX_BUFFSIZE 255
void main()
{
    char *str = "This is a test.";
    char outBuf[MAX_BUFFSIZE];
    int len;

    stdio_init_all();

    while( 1 )
    {
        len = mytoupper( str, outBuf );
        printf("Before str: %s\n", str);
        printf("After str: %s\n", outBuf);
        printf("Str len = %d\n", len);
    }
}
```

We changed the name of our toupper function to **mytoupper**, since there is already a toupper function in the C runtime. Without this change, there is a multiple definition error. This was done in both the C and the Assembly Language code; otherwise, the function is the same as in Chapter 7. The **CMakeLists.txt** file is as expected simply listing both **upper.S** and **uppertst.c**.

Define the parameters and return code for our function to the C compiler. We do this with

```
extern int mytoupper( char *, char * );
```

This should be familiar to all C programmers, as you must do this for C functions as well. Usually, you gather up all these definitions and put them in a header **(**.h**)** file.

When the program is run, the string is in uppercase as expected, but the string length appears one greater than anticipated. That is because the length includes the NULL character, which isn't the C standard. If we really wanted to use this a lot with C, subtract 1 so that our length is consistent with other C runtime routines.

How to Embed Assembly Code Inside C Code

The GNU C compiler allows Assembly Language code to be embedded in the middle of C code. It contains features to interact with C variables and labels and cooperate with the C compiler and optimizer for register usage. Listing 8-5 is a simple example, where we embed the core algorithm for the toupper function inside the C program.

Listing 8-5. Embedding Our Assembly Routine Directly in C Code

```
//
// C program to embed our Assembly Language
// toupper routine inline.
//

#include <stdio.h>
#include "pico/stdlib.h"

#define MAX_BUFFSIZE 255
void main()
{
    char *str = "This is a test.";
    char outBuf[MAX_BUFFSIZE];
    int len;
```

```
    stdio_init_all();

    while( 1 )
    {
        asm
        (
            "MOV R0, %1\n"
            "MOV R4, %2\n"
            "loop: LDRB R5, [R0]\n"
            "ADD R0, #1\n"
            "CMP R5, #'z'\n"
            "BGT cont\n"
            "CMP R5, #'a'\n"
            "BLT cont\n"
            "SUB R5, #('a'-'A')\n"
            "cont: STRB R5, [%2]\n"
            "ADD %2, #1\n"
            "CMP R5, #0\n"
            "BNE loop\n"
            "SUB R0, %2, R4\n"
            "MOV %0, R0\n"
            "MOV %2, R4"
            : "=r" (len)
            : "r" (str), "r" (outBuf)
            : "r4", "r5", "r0"
        );

        printf("Before str: %s\n", str);
        printf("After str: %s\n", outBuf);
        printf("Str len = %d\n", len);
    }
}
```


The **asm** statement allows Assembly Language code to be embedded directly into C code. Having done this, we could write an arbitrary mixture of C and Assembly Language. The comments are stripped out from the Assembly Language code, so the structure of the C and Assembly Language is easier to read. The general form of the **asm** statement is

```
asm asm-qualifiers ( AssemblerTemplate
: OutputOperands
[ : InputOperands]
[ : Clobbers ] ]
[ : GotoLabels])
```

The parameters are

- **AssemblerTemplate**: A C string containing the Assembly code. There are macro substitutions that start with % to let the C compiler insert the inputs and outputs.
- **OutputOperands**: A list of variables or registers returned from the code. This is required, since it is expected that the routine does something. In this case, this is **"=r" (len)** where the **=r** means an output register and that it goes into the C variable **len**.
- **InputOperands**: A list of input variables or registers used by our routine, in this case "r" (str); "r" (outBuf) means we want two registers: one holds **str** and one holds **outBuf.** It is fortunate that C string variables hold the address of the string, which is what is wanted in the register. These registers need to be preserved. The C compiler expects them to be unchanged once the code exits and any changes cause bugs.

- **Clobbers**: A list of registers used and clobbered when the code runs, in this case "**r0**," "**r4**," and "**r5**."
- **GotoLabels**: A list of C program labels that the code might want to jump to. Usually, this is an error exit. If you jump to a C label, warn the compiler with a **goto asm-qualifier**.

You can label the input and output operands, which we didn't, and that means the compiler will assign names **%0, %1**, … as used in the Assembly Language code.

If the program is disassembled, you will find that the C compiler avoids using registers **R0**, **R4**, and **R5** entirely, leaving them open to use. It loads input registers from the variables on the stack, before the code executes, and then copies a return value from the assigned register to the variable **len** on the stack. It doesn't give the same registers originally used, but that isn't a problem.

The input registers for **instr** and **oustr** can't be modified. For **outstr**, since its value was saved to **R4** for the length calculation, we can restore that at the end. We move **instr** into **R0** and increment that so that the input register is preserved.

Note If you have too many registers specified, then you may receive your inputs in high registers. How data is moved in and out of the lower registers for processing needs to be managed. In the case of this program, it is fine when built for debug, but when built for nodebug, %0 ends up in **R8**. This is why the final subtraction is to **R0**, and then that is moved to %0.

This routine is straightforward and doesn't have any ill side effects. If the Assembly Language code is accessing hardware registers, add a volatile keyword to the **asm** statement to make the C compiler more conservative

on any assumptions it makes about the code. Otherwise, the C compiler doesn't know hardware registers can change independently from your code and the optimizer might remove important code.

Summary

In this chapter, we studied calling C functions from Assembly Language code. We used functions in the RP2040's SDK to access the GPIO pins and noted how to deal with inline C functions. We then did the reverse and called the Assembly Language uppercase function from a C main program. Next, we embedded Assembly Language code directly inline into C code.

Accessing the RP2040's hardware indirectly through the SDK works and is quick, but as Assembly Language programmers, we like to access the hardware directly, which is the topic of Chapter 9.

Exercises

8-1. Create a C program to call the lowercase routine from Exercise 7-2, and print out some test cases.

8-2. Take the lowercase routine from Exercise 7-2, and embed it in C code using an **asm** statement.

8-3. Review the main routine in the **.dis** file for the embedded Assembly Language. See how the main routine C code is converted to Assembly Language, saves the registers, creates a stack frame, and passes the addresses of **instr** and **outstr**.

8-4. Modify the flashing lights program to flash the lights in different patterns and vary the sleep times. Would this be easier if the handling of each LED was moved into a function?

CHAPTER 9

How to Program the Built-in Hardware

In Chapter 8, we interacted with external hardware devices connected to the GPIO pins using the RP2040's SDK. In this chapter, we look at interacting with the hardware directly. To do this, we don't need to learn any additional Assembly Language instructions because we access the hardware with the memory **load/store** instructions we previously studied. All hardware access is via special memory addresses connected to hardware devices that respond based on the data written to them rather than being connected to memory. Similarly, hardware devices provide data from external sources when these addresses are read.

Before delving into individual registers directly, we need a lay of the land. This chapter gives details about the RP2040's memory map.

About the RP2040 Memory Map

The RP2040 contains several types of memory plus a large selection of hardware registers:

- Two banks of read-only memory
- The 264KB of read-write memory
- Several large banks of hardware registers that control the hardware or send/receive data to/from it

S. Smith, *RP2040 Assembly Language Programming*,
https://doi.org/10.1007/978-1-4842-7753-9_9

Table 9-1 is a high-level map of the main memory areas.

***Table 9-1.** High-Level Memory Map of the RP2040*

Base address	Purpose
0x00000000	On-chip 16KB Boot ROM
0x10000000	Off-chip flash memory 16MB Max, RP Pico has 2MB
0x20000000	On-chip SRAM 264KB partitioned into six banks
0x40000000	Hardware registers for peripherals connected to the APB Bridge
0x50000000	Hardware registers for devices connected to AHB Bus
0xd0000000	Hardware registers connected directly to CPU such as SIO
0xe0000000	Arm Cortex-M0+ processor hardware registers

When we looked at the disassembly for one of the programs, all the code addresses were in the 0x10000000 range, indicating the program is running from the Pico's ROM. This preserves our program between power resets and is what the 16KB boot loader will run on power-up. The data variables and the stack are in the 0x20000000 range, indicating these aren't stored over power resets but are easy to write to. As we proceed with studying the RP2040, we'll use registers from these various sets. This is how the programs view the various hardware devices connected to the RP2040. Next, we look at referring to these memory addresses and registers in a friendlier manner.

About C Header Files

It is poor programming to use magic numbers in code. Therefore, when programming the SIO pins, don't just plunk the number 0xd0000000 in the code; instead, use a symbolic reference. We don't need to define these using **.EQU** statements, as these are all defined in the SDK. For instance, 0xd0000000 is defined in src/rp2040/hardware_regs/include/hardware/regs/addressmap.h with

```
#define SIO_BASE _u(0xd0000000)
```

The file **addressmap.h** is a C header file, and **#define** is a C preprocessor definition. The C preprocessor replaces SIO_BASE with _u(0xd0000000) everywhere before compiling the source code. But aren't we programming in Assembly Language? How can we use C header files?

This is why the source files are named with an uppercase **.S** extension. The **.S** instructs the GNU Assembler to accept and process C source files. If a lowercase **.s** extension is used, then the GNU Assembler accepts strict Assembly Language and spits out lots of error messages. The C header file must be a simple set of defines to work; if it defines C functions or structures, then the resulting code won't compile.

The designers of the RP2040 SDK kept Assembly Language programmers in mind when defining header files; header files can be safely included for the various memory locations and values of all the hundreds of hardware memory registers.

In this case, the SIO_BASE definition is used with

```
gpiobase: .word SIO_BASE  @ base of the GPIO registers
```

Note The name is SIO_BASE rather than GPIO_BASE to emphasize programming through the single-cycle IO controller. We'll see how this helps us shortly.

These are the basics for programming access. Next, we connect hardware devices to the outside world via the pins exposed on the boards, specifically to the Raspberry Pi Pico. For directions on how to connect other manufacturer's RP2040 boards, refer to their documentation.

About the Raspberry Pi Pico Pins

If you observe the Raspberry Pi Pico's external pins, you see that each pin is labeled with several functions. The various peripherals contained in the RP2040 are connected to the external pins through the Advanced Peripheral Bus (APB). The APB has a programmable multiplexor where each peripheral is specified to connect to each pin. Each pin can be programmed to do one of up to nine functions. Which nine functions are possible for each pin is hard-coded in the hardware, but much flexibility is allowed in designing projects.

Note The ground and power pins are fixed and not connected to the APB.

For example, for GPIO pins 18, 19, and 20 that were connected to LEDS in Chapter 8, Table 9-2 lists their other available functions.

Table 9-2. *Functions for Pins 18, 19, and 20*

Pin	F1	F2	F3	F4	F5	F6	F7	F8	F9
18	SPI0 SCK	UART0 CTS	I2C1 SDA	PWM1 A	SIO	PIO0	PIO1		USB OVCUR DET
19	SPI0 TX	UART0 RTS	I2C1 SCL	PWM1 B	SIO	PIO0	PIO1		USB VBUS DET
20	SPI0 RX	UART1 TX	I2C0 SDA	PWM2 A	SIO	PIO0	PIO1	CLOCK GPIN0	USB VBUS EN

Table 9-3 lists the hardware functions with a quick description of their purpose.

Table 9-3. *Description of Hardware Peripheral Functions*

Peripheral	Description
SPI	Serial Peripheral Interface. A synchronous serial communication interface specification used for short-distance communication
UART	Universal Asynchronous Receiver/Transmitter. For asynchronous serial communication in which the transmission speeds are configurable
I2C	Inter-Integrated Circuit. A synchronous, multimaster, multislave, packet switched, single-ended, serial communication bus
PWM	Pulse-Width Modulation. A method of reducing the average power delivered by an electrical signal by turning on and off with a variable pulse width. Commonly used to control motors
SIO	Single-cycle IO. Software control of GPIO pins
PIO	Programmable IO. Connected to one of the PIO coprocessors
CLOCK GPIN	General-purpose clock inputs. Can be routed to a number of internal clock domains on RP2040
CLOCK GPOUT	General-purpose clock outputs. Can drive a number of internal clocks onto external pins
USB OVCUR	USB power control signals to/from the internal USB controller

To flash the LEDs, first set the function of pins 18, 19, and 20 to SIO so the program can control them.

How to Set a Pin Function

To configure a pin as a general-purpose programmable pin, set a hardware register to program the APB to route SIO functionality to the external pin. The addresses of all the various banks of hardware registers are defined in **addressmap.h**. The **define** to use is

```
#define IO_BANK0_BASE _u(0x40014000)
```


For each pin, there are two 32-bit registers:

- Status register
- Control register

This means to access the register

1. Multiply the pin number by 8. Multiply by 8 by shifting the pin number left by 3 bits and then add that to the base.
2. Add that to the base to get the registers for the desired pint. This gives us the address of the set of registers for the target pin.
3. Access the control register by providing the offset IO_BANK0_GPIO0_CTRL_OFFSET, from io_bank0.h, to the **STR** instruction.
4. To configure the APB write 5 to the control register, instead of 5, use the constant IO_BANK0_GPIO3_CTRL_FUNCSEL_VALUE_SIO_3 from io_bank0.h.

The code to do this follows in Listing 9-1.

Listing 9-1. Code to Set the GPIO Pin to the SIO Function, Where the Pin Is Provided in R0

```
#include "hardware/regs/addressmap.h"
#include "hardware/regs/io_bank0.h"
      LDR R2, iobank0    @ address we want
      LSL R0, #3         @ each GPIO has 8 bytes of registers
      ADD R2, R0         @ add the offset for the pin number
      MOV R1, #IO_BANK0_GPIO3_CTRL_FUNCSEL_VALUE_SIO_3
      STR R1, [R2, #IO_BANK0_GPIO0_CTRL_OFFSET]
...
iobank0: .WORD IO_BANK0_BASE @ base of io config registers
```

Note **iobank0** must be defined in the code section, not the data section, so it can be loaded with one **LDR** instruction.

Programming this control register is easy since only a value is required to be written to it. This isn't true, in general, and the RP2040 provides help to make programming hardware registers easier, which is shown next.

About Hardware Registers and Concurrency

Most hardware registers are 32 bits, and each bit performs a different function. For instance, the register to turn on and off the GPIO pins has all the external pins in one register, and to set or clear pins, be careful not to mess with other bits. The logic to do this would resemble

```
LDR R1, [R2]   @ R2 is the address of the hardware register
ORR R1, R3     @ R3 has one bit set that we want to effect
STR R1, [R2]   @ Write the value back to the register with one
                 bit altered
```

There are problems with this; besides taking three instructions and, perhaps, being error prone, the big problem is concurrency. The RP2040 has two CPU cores, so separate functions could run on each CPU core performing different operations on different SIO pins.

If one CPU does the **LDR** but then the other CPU does the **LDR** before the first CPU does the **STR**, then the second CPU will undo what the first CPU does when it performs its **STR** instruction, as shown in Figure 9-1.

CPU 1	CPU 2	Comment
LDR R1, [R2]		
	LDR R1, [R2]	Both CPUs have read the same value of the register
ORR R1, R3	ORR R1, R3	Both CPUs set their separate bits
STR R1, [R2]		CPU1 writes back what it wants
	STR R1, [R2]	CPU2 overwrites CPU1's work

Figure 9-1. *Flow of two CPUs with a concurrency problem*

The RP2040 solves this problem by having separate registers for performing different operations on the registers. In the case of setting or clearing SIO pins, there are two registers:

- **One to set the pins**: To set one or more pins, you use the SET register. Each bit is for a different pin. You just write a value to the set register, where any one bit in your value will turn on that SIO pin. Any zero bits written are ignored, and those pins are left alone.
- **One to clear the pins**: To clear pins, there is a clear (CLR) register where any 1 bit will clear a GPIO pin and again zeros are ignored.

This scheme is why the name SIO for single-cycle I/O, since we only need one instruction; thus, one clock cycle sets or clears an I/O pin. On some pins, there is also an **XOR** register that only sets the value if the pin isn't already set, perhaps saving the hardware work. These registers are laid out in two patterns:

1. For Raspberry designed devices like SIO, they are in consecutive registers, where each one is defined in a header file.
2. For devices taken from an ARM chip design library, Raspberry provides aliases to the ARM defined registers. You usually access the single-cycle register by setting a bit in the defined address of the register. These bits are defined in **addressmap.h** starting with REG_ALIAS; an example of this is provided when configuring the pin's external pad.

After the function of the pins is programmed, the pads must be initialized.

About Programming the Pads

The APB is connected to the outside world with **pads**. Pads provide electrical isolation and control voltage and current levels. Program these to turn them on, for both input and output. In this chapter, instructions for programming output are given, but it doesn't hurt to turn both on. Strangely enough, input is turned on with input enable; however, turning off the output with output disable means only setting the input enable bit to configure the pad, as follows in Listing 9-2.

Listing 9-2. How to Configure a Pad

```
LDR R2, padsbank0
LSL R3, R0, #2     @ pin * 4 for register address
ADD R2, R3         @ Actual set of registers for pin
MOV R1, #PADS_BANK0_GPIO0_IE_BITS
LDR R4, setoffset
ORR R2, R4
STR R1, [R2, #PADS_BANK0_GPIO0_OFFSET]
...
padsbank0: .word PADS_BANK0_BASE
setoffset: .word REG_ALIAS_SET_BITS
```

Notice how the address of padsbank0 is loaded, to add in the offset for the GPIO pin desired; then ORR with the bit gives the alias to the SET single-cycle register.

How to Initialize SIO

In this next step, the SIO device is initialized, preparing the pin for output and turning it off (in case it was previously turned on). There are 26 pins exposed externally—pins 0 to 28, excluding 23 to 25. They can each be referenced by a bit in a 32-bit register. Access that bit by placing a one in a register and shifting it left by the pin number.

To initialize the SIO pin

1. Write one to the pin's position in the output enable set register to configure it for output.
2. Write the same value to the output clear register to turn the pin off.

Listing 9-3 shows this process.

Listing 9-3. How to Configure the SIO Pin to a Known State

```
#include "hardware/regs/addressmap.h"
#include "hardware/regs/sio.h"
...
    MOV R3, #1
    LSL R3, R0                @ shift over to pin position
    LDR R2, gpiobase          @ address we want
    STR R3, [R2, #SIO_GPIO_OE_SET_OFFSET]
    STR R3, [R2, #SIO_GPIO_OUT_CLR_OFFSET]
...
gpiobase: .WORD SIO_BASE   @ base of the GPIO registers
```

How to Turn a Pin On/Off

To turn on a pin is the same process as before, except now write it to the SIO set register to turn on the current to drive the LED as shown in Listing 9-4.

Listing 9-4. Code to Turn On a LED by Turning On the SIO Output Register

```
MOV R3, #1
LSL R3, R0         @ shift over to pin position
LDR R2, gpiobase   @ address we want
STR R3, [R2, #SIO_GPIO_OUT_SET_OFFSET]
```

Similarly, turn the LED off by doing the same thing to the SIO clear register.

Note It takes only one instruction to access the SIO, adding efficiency, simplifying programming, and eliminating concurrency problems.

The Complete Program

Putting all the program together is shown in Listing 9-5. This program uses the good programming practice of employing constants in the C header files. The program demonstrates using hardware registers. It doesn't use the SDK to access the SIO pins; instead, it only uses the SDK for the **sleep_ms** function.

Listing 9-5. The Complete Program to Flash the LEDs Writing to the Hardware Directly

```
@
@ Assembler program to flash three LEDs connected to the
@ Raspberry Pi GPIO writing to the registers directly.
@
@

#include "hardware/regs/addressmap.h"
#include "hardware/regs/sio.h"
#include "hardware/regs/io_bank0.h"
#include "hardware/regs/pads_bank0.h"

     .EQU LED_PIN1, 18
     .EQU LED_PIN2, 19
     .EQU LED_PIN3, 20

     .EQU sleep_time, 200
```



```
.thumb_func
.global main                 @ Provide program starting address

     .align  4               @ necessary alignment
main:
@ Init each of the three pins and set them to output
     MOV R0, #LED_PIN1
     BL  gpioinit
     MOV R0, #LED_PIN2
     BL  gpioinit
     MOV R0, #LED_PIN3
     BL  gpioinit

loop:
@ Turn each pin on, sleep then turn the pin off
     MOV R0, #LED_PIN1
     BL  gpio_on
     LDR R0, =sleep_time
     BL  sleep_ms
     MOV R0, #LED_PIN1
     BL  gpio_off
     MOV R0, #LED_PIN2
     BL  gpio_on
     LDR R0, =sleep_time
     BL  sleep_ms
     MOV R0, #LED_PIN2
     BL  gpio_off
     MOV R0, #LED_PIN3
     BL  gpio_on
     LDR R0, =sleep_time
     BL  sleep_ms
     MOV R0, #LED_PIN3
     BL  gpio_off
```

```
    B   loop                @ loop forever

@ Initialize the GPIO to SIO. r0 = pin to init.
gpioinit:
@ Initialize the GPIO
    MOV R3, #1
    LSL R3, R0              @ shift over to pin position
    LDR R2, gpiobase        @ address we want
    STR R3, [R2, #SIO_GPIO_OE_SET_OFFSET]
    STR R3, [R2, #SIO_GPIO_OUT_CLR_OFFSET]

@ Enable input and output for the pin
    LDR R2, padsbank0
    LSL R3, R0, #2          @ pin * 4 for register address
    ADD R2, R3              @ Actual set of registers
                              for pin
    MOV R1, #PADS_BANK0_GPIO0_IE_BITS
    LDR R4, setoffset
    ORR R2, R4
    STR R1, [R2, #PADS_BANK0_GPIO0_OFFSET]

@ Set the function number to SIO.
    LSL R0, #3              @ each GPIO has 8 bytes of
registers
    LDR R2, iobank0         @ address we want
    ADD R2, R0              @ add the offset for the pin
                              number
    MOV R1, #IO_BANK0_GPIO3_CTRL_FUNCSEL_VALUE_SIO_3
    STR R1, [R2, #IO_BANK0_GPIO0_CTRL_OFFSET]
    BX  LR
```



```
@ Turn on a GPIO pin.
gpio_on:
    MOV R3, #1
    LSL R3, R0              @ shift over to pin position
    LDR R2, gpiobase        @ address we want
    STR R3, [R2, #SIO_GPIO_OUT_SET_OFFSET]
    BX  LR

@ Turn off a GPIO pin.
gpio_off:
    MOV R3, #1
    LSL R3, R0              @ shift over to pin position
    LDR R2, gpiobase        @ address we want
    STR R3, [R2, #SIO_GPIO_OUT_CLR_OFFSET]
    BX  LR

        .align  4           @ necessary alignment
gpiobase: .word SIO_BASE    @ base of the GPIO registers
iobank0: .word IO_BANK0_BASE    @ base of io config registers
padsbank0: .word PADS_BANK0_BASE
setoffset: .word REG_ALIAS_SET_BITS
```

The SDK gpio_init function defaults setting the SIO pin for input, so we needed to call gpio_set_dir to set the pin for output. In this example, the **gpioinit** function sets the pin for output, so the extra function isn't required.

Summary

In this chapter, we studied how the memory in the RP2040 is organized, where ROM and RAM and where the hardware registers are located. We learned how to use the C header files in the SDK to get symbolic references for the hardware registers and their values. We then studied how the internal hardware devices are connected to external pads that we soldered pins to. We programmed the APB and pins to connect and make the SIO pins we wished to use active. We then configured the SIO pins to turn them on and off. To conclude, an Assembly Language version of the Chapter 8 program was written, that writes to the hardware directly rather than using the SDK functions.

This method of accessing the hardware is called "bit banging," where one CPU bangs the bits in the hardware registers to do what is wanted. This method is expensive on the ARM Cortex-M0+'s processor. In Chapter 10, we learn to offload this work to the RP2040's I/O coprocessors in order to free up the ARM CPU for other useful work.

Exercises

9-1. What is the starting memory address for the hardware registers for I2C number 0 I/O device? Which header file do we look in for useful defines when working with this device?

9-2. Why does the Raspberry Pi Pico have multiple functions on each external pin? Why doesn't the Pico just have more pins so you can use them all at once?

9-3. Try changing the program to flash the LEDs in a different pattern. Can you add a fourth and fifth LED?

9-4. To make sure you understand how the program loads the hardware addresses, single-step through the program to examine how addresses are loaded step by step. Look at the disassembly file to see what the code is assembled into.

9-5. How would you structure the program to do other work, rather than calling **sleep_ms()**?

CHAPTER 10

How to Initialize and Interact with Programmable I/O

So far, we've studied the Assembly Language instructions for the ARM Cortex-M0+ processor. In this chapter, we put that aside and look at a new Assembly Language syntax quite different from ARM's. The RP2040 contains eight programmable I/O (PIO) processors that are programmed as state machines with their own Assembly Language instructions. There is a tool in the SDK, pioasm, which assembles these in a similar manner to the GNU Assembler we have used.

The RP2040 contains several specialized I/O hardware components for handling various common hardware protocols like the UART and USB. However, with DIY projects, you often encounter nonstandard devices that require custom control of the GPIO pins. Sometimes, it is possible to implement these protocols using the ARM CPU in a manner similar to that in Chapter 9, but the ARM CPU wasn't designed for this, and it takes all the ARM's processing power if it is even possible. Raspberry's solution to this is the PIO processors that offload the processing from the CPU and hopefully provide enough programming power to accomplish most common jobs. Controlling I/O isn't an easy job, but it isn't necessary to design custom hardware or add a second RP2040 board to perform the I/O.

S. Smith, *RP2040 Assembly Language Programming*,
https://doi.org/10.1007/978-1-4842-7753-9_10

The good news is that we only need to learn nine Assembly Language instructions, and there are only 32-instruction memory slots shared by four PIO processors. Each instruction executes in one clock cycle and sets or reads a set of GPIO pins, meaning we can manage protocols that operate up to 125MHz. This excludes HDMI but encompasses most other things including VGA. The trick is how to implement protocols in small compact programs that don't stall waiting for some external event.

Before diving into an example, we first look at the architecture of the PIO system.

About PIO Architecture

There are eight PIO coprocessors that are divided into two banks of four. Each bank of four shares the same 32-instruction memory for program storage. Figure 10-1 is a block diagram of one of the PIO coprocessors.

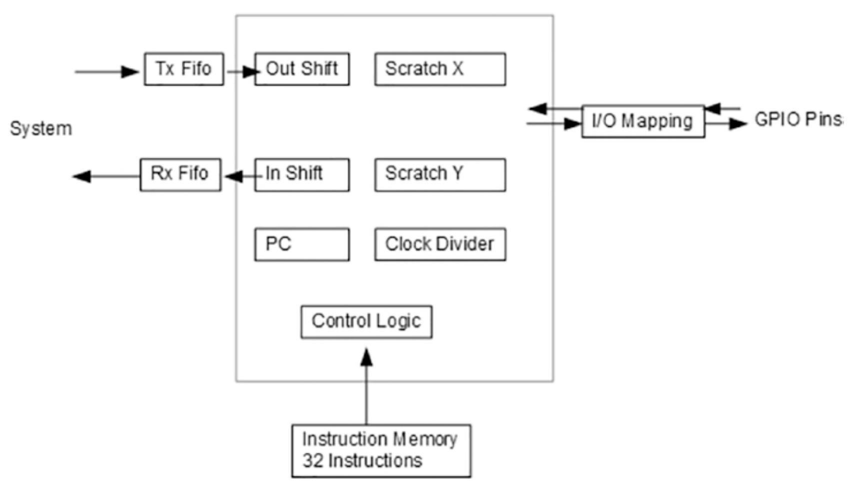

Figure 10-1. *Block diagram of one PIO processor*

Within each PIO, there are

- Two general-purpose 32-bit scratch registers
- Two shift registers to assist in shifting bits into and out of the processor
- A four-word transmit FIFO to buffer data coming from the ARM CPU
- A four-word receive FIFO to buffer data being sent to the ARM CPU
- A program counter that controls which instruction is being executed
- A clock divider register that slows down PIO processing
- The I/O mapping that maps the PIO output to physical GPIO pins
- The control logic that executes the instructions

Each instruction is 16 bits in length and comprised of three parts:

1. The operand is like the operands we used from the ARM world.
2. A side-set value set to the configured side-set pins. This means every instruction can change the GPIO pins for fastest processing.
3. A delay value which slows an instruction up to 31 clock cycles to help program precise timing to match hardware protocol requirements.

Note Besides the delay value, the overall program can be slowed by setting the clock divider register.

Next, we look at the nine individual instructions.

About the PIO Instructions

In this section, we look at nine instructions and their operands. All these instructions can have a side-set or delay value included, but for simplicity, we look at that in the following sections.

1. JMP condition address
2. WAIT polarity source index
3. IN source, bitcount
4. OUT destination, bitcount
5. PUSH if-full block
6. PULL if-empty block
7. MOV destination, operation source
8. IRQ set/wait irq_num _rel
9. SET destination, value

Four of the instructions—IN, OUT, PUSH, and PULL—are concerned with transferring data to and from the ARM CPU. There aren't any memory operations, and the arithmetic operations are limited. The JMP instruction can decrement a counter, and the MOV instruction can reverse the bits or perform a one's complement as part of the move.

Before we go into detail on these instructions, an example follows to get a feel for how these instructions are used.

Flashing the LEDs with PIO

We flashed three LEDs with the SDK, writing directly to the RP2040's hardware registers and now using the PIO coprocessor. The advantage to this method is that all the processing happens on three PIOs and the ARM processor is left free to do other work. We'll start with the PIO Assembly Language code and put it in a file called **blink.pio** containing Listing 10-1.

Listing 10-1. PIO Assembly Language Code to Blink a LED

```
;
; Program to blink a LED
;

.program blink
    pull block
    out y, 32
.wrap_target
    mov x, y
    set pins, 1   ; Turn LED on
lp1:
    jmp x-- lp1   ; Delay for (x+1) cycles, x is a 32 bit
number
    mov x, y
    set pins, 0   ; Turn LED off
lp2:
    jmp x-- lp2   ; Delay for the same number of cycles again
    mov x, y
lp3:              ; Do it twice to wait for 2 other leds to
blink
    jmp x-- lp3   ; Delay for the same number of cycles again
.wrap             ; Blink forever!
```

```
% c-sdk {
// this is a raw helper function for use by the user which sets
// up the GPIO output, and configures the SM to output on a
// particular pin

void blink_program_init(PIO pio, uint sm, uint offset, uint
pin) {
   pio_gpio_init(pio, pin);
   pio_sm_set_consecutive_pindirs(pio, sm, pin, 1, true);
   pio_sm_config c = blink_program_get_default_config(offset);
   sm_config_set_set_pins(&c, pin, 1);
   pio_sm_init(pio, sm, offset, &c);
}
%}
```

First a few notes about this file:

- Comments start with a semicolon, anything after a semicolon is ignored. C style comments /* */ and // can also be used.
- The program starts with a .program directive that gives the program a name. This will be used in C variable names, so the rules for a C variable must be followed.
- The PC wraps back to 0 once it passes 31, giving an infinite loop for free. However, there are control registers that can alter this wrap around, namely, setting the end instruction and then where to loop to. The **.wrap** and **.wrap_target** directives define this setting to give an infinite loop, saving the use of an extra **JMP** instruction.
- Labels are like ARM Assembly, a name followed by a colon. These are used as the targets for **JMP** instructions.

- This file will be assembled into a C header (.h) file containing the machine code 16-bit instructions in an array. As a consequence, we can include C code in this file, where anything between % c-sdk { and %} is put in the resulting header file along with a couple of other generated helper functions.

The program inputs a 32-bit delay loop counter from the ARM world and keeps that in the **Y** scratch register, and whenever it needs to wait, it moves this to the **X** scratch register and then loops that many times. The program turns on the LED, does the delay loop, then turns the LED off. It then performs the delay loop twice to let the other two LEDs have their turn. Which pin the program controls is configured from the ARM side. Here's a quick overview of what each instruction does:

1. **Pull block**: Pulls a 32-bit quantity from the host Tx FIFO into the output shift register (OSR). The block operand says to wait for a quantity.
2. **Out y, 32**: Shifts 32 bits from the OSR into the Y scratch register.
3. **Mov x, y**: Copies the contents of the Y scratch register to the X scratch register.
4. **Set pins, 1**: Sets the pins configured for this PIO to 1. The pin to use is configured by the C program.
5. **Jmp x-- lp1**: Jumps to lp1 if X is nonzero while decrementing the X scratch register. The condition is based on the initial value of X.
6. **Set pins, 0**: Turns off the pins configured for this PIO.

Although the PIOs do all the work, a C (or ARM Assembly Language) program must download the code to the PIOs, configure them, and send the loop count in. This is done by the program **blink.c** containing Listing 10-2.

Listing 10-2. The C Code to Call the SDK to Download and Configure the PIOs

```
/**
 * C Program to set the PIO in motion blinking the LEDs
 */

#include <stdio.h>

#include "pico/stdlib.h"
#include "hardware/pio.h"
#include "hardware/clocks.h"
#include "blink.pio.h"

const uint LED_PIN1 = 18;
const uint LED_PIN2 = 19;
const uint LED_PIN3 = 20;

#define SLEEP_TIME 200

void blink_pin_forever(PIO pio, uint sm, uint offset, uint pin,
uint freq);

int main() {
    int i = 0;

    setup_default_uart();

    PIO pio = pio0;
    uint offset = pio_add_program(pio, &blink_program);
    printf("Loaded program at %d\n", offset);
```



```
    blink_pin_forever(pio, 0, offset, LED_PIN1, 5);
    sleep_ms(SLEEP_TIME);
    blink_pin_forever(pio, 1, offset, LED_PIN2, 5);
    sleep_ms(SLEEP_TIME);
    blink_pin_forever(pio, 2, offset, LED_PIN3, 5);
    while(1)
    {
        i++;
        printf("Busy counting away i = %d\n", i);
    }
}

void blink_pin_forever(PIO pio, uint sm, uint offset,
              uint pin, uint freq) {
    blink_program_init(pio, sm, offset, pin);
    pio_sm_set_enabled(pio, sm, true);

    printf("Blinking pin %d at %d Hz\n", pin, freq);
    pio->txf[sm] = clock_get_hz(clk_sys) / freq;
}
```

The C program uses three PIO processors in PIO bank 0. There are two banks of four PIOs, where each bank shares the same 32-instruction memory. It downloads the program using the *pio_add_program* SDK function. The program is contained in **blink_pio.h** as a 16-bit unsigned integer array containing comments showing how each instruction was assembled:

```
static const uint16_t blink_program_instructions[] = {
    0x80a0, //  0: pull   block
    0x6040, //  1: out    y, 32
            //     .wrap_target
    0xa022, //  2: mov    x, y
    0xe001, //  3: set    pins, 1
```

```
    0x0044, //  4: jmp    x--, 4
    0xa022, //  5: mov    x, y
    0xe000, //  6: set    pins, 0
    0x0047, //  7: jmp    x--, 7
    0xa022, //  8: mov    x, y
    0x0049, //  9: jmp    x--, 9
            //     .wrap
};
```

Next, the program starts each PIO, sleeping 200ms between so that each one blinks at the correct time. Once the PIOs are set in motion, the C program that runs on the ARM CPU goes into an infinite loop printing a count. This demonstrates that the ARM CPUs are both completely free to do other work, while the three PIO processors flash the LEDs.

To assemble the PIO code, add a line to the **CMakeLists.txt** file as shown in Listing 10-3 where a **pico_generate_pio_header** statement is added.

Listing 10-3. CMakeLists.txt File with pico_generate_pio_header statement

```
cmake_minimum_required(VERSION 3.13)

include(pico_sdk_import.cmake)
project(test_project C CXX ASM)

set(CMAKE_C_STANDARD 11)
set(CMAKE_CXX_STANDARD 17)

pico_sdk_init()

add_executable(pio_blink)

# by default the header is generated into the build dir
pico_generate_pio_header(pio_blink ${CMAKE_CURRENT_LIST_DIR}/
blink.pio)
```



```
target_sources(pio_blink PRIVATE blink.c)

target_link_libraries(pio_blink PRIVATE pico_stdlib hardware_pio)
pico_add_extra_outputs(pio_blink)
```

The C code that calls SDK functions to control the PIOs is standard and taken from the various PIO samples included in the SDK. As more sophisticated programs are developed, we'll discuss how these need to be modified, but first, we look at the individual PIO instructions in more detail.

PIO Instruction Details and Examples

Each instruction is simple, but they have many variations. In this section, examples of each instruction are given in its various forms.

JMP

The PIO doesn't have a program status register, so the conditions are based on various operations in the PIO. Here are all the incarnations of the **JMP** instruction:

```
JMP label ; unconditional branch
JMP !X label      ; jump if X is non zero
JMP X—label       ; jump if X is nonzero while decrementing X
JMP !Y label      ; jump if Y is non zero
JMP Y—label       ; jump if Y is non zero while decrementing Y
JMP X!=Y label    ; jump if X is not equal to Y
JMP pin label     ; jump if pin is 1
JMP !OSRE label   ; jump if the OSR has less bits
                  ; than the configured threshold
```

Note The **pin** and **!OSRE** versions of jump require configuration from the SDK function sm_config_set_jmp_pin or sm_config_set_out_shift.

WAIT

WAIT can wait for a source to be 0 or 1 based on its first polarity instruction. Here are examples with each source:

```
WAIT 0 gpio 17      ; wait for GPIO 17 to be 0 (actual GPIO pin)
WAIT 1 pin 1        ; wait for pin 1 to be 1 (mapped pins)
WAIT 1 irq 1        ; wait for IRQ 1 to be set (then clears it)
WAIT 0 irq 2 rel    ; wait for IRQ 2 to clear,
                    ; IRQ is relative to other PIOs.
```

Interrupts are discussed in Chapter 11. The other two forms let us wait on a physical GPIO with the **gpio** version or wait on a configured pin with the **pin** version.

IN

When performing I/O, usually bits are received one by one. The purpose of the input shift register (ISR) is to accumulate these bits one by one, and when there's a byte or word, those are sent to the ARM CPU. The IN instruction moves bits from one of various sources into the ISR. Here are all the forms of the IN instruction:

```
IN PINS, 1    ; Move 1 bit from the configured pins to the ISR
IN X, 32      ; Copy the entire X register to the ISR
IN Y, 16      ; Copy 16 bits from the Y register to the ISR
```



```
IN NULL, 4    ; Copy 4 zero bits into the ISR
IN ISR, 4     ; Can be used to rotate 4 bits in the ISR
IN OSR, 8     ; Copy 8 bits from the OSR to the ISR
```

Transferring data is straightforward.

OUT

OUT transfers bits from the output shift register into various destinations inside the PIO. This data is received from the ARM CPU, which was already moved from the transmit FIFO into the OSR. Here are the forms of the OUT instruction:

```
OUT PINS, 1       ; set the pins from one bit in the OSR
OUT X, 32         ; move 32 bits from the OSR to the X register
OUT Y, 8          ; move one byte from the OSR to
                  ; the Y register
OUT NULL, 16      ; delete 16 bits from the OSR
OUT PINDIRS, 1    ; sets the pin direction for the mapped pins
OUT PC, 5         ; jump to the alocation in the
                  ; next 5 bits of the OSR
OUT ISR, 16       ; move 16 bits to the ISR
OUT EXEC, 16      ; execute the next 16 bits as an instruction
```

OUT is the reverse of IN, except that it controls the direction of the pins in a couple of interesting ways, including the host controlling the PIO by copying data to the PC to perform a jump or using **EXEC** to execute single instructions.

PUSH

PUSH pushes the contents of the ISR into the Rx FIFO as a single 32-bit quantity and then sets the ISR to 0. **PUSH** blocks if the Rx FIFO is full, or if **noblock** is set, then **PUSH** continues to the next instruction without doing anything. The **ifful** parameter tells PUSH not to do anything, unless the ISR has reached a certain threshold of bits received.

```
PUSH block           ; Push the ISR to the Rx FIFO waiting
                     ; for space to be available
PUSH noblock         ; Push the ISR to the Rx FIFO if
                     ; space available else no-op
PUSH iffull block    ; Push ISR to Rx FIFO if enough bits
                     ; received and space available
PUSH iffull noblock  ; Push ISR to Rx FIFO if enough bits
                     ; received and space available, else no-op
```

Note There is an **autopush** configuration that pushes automatically without requiring this instruction.

PULL

PULL pulls a 32-bit quantity from the Tx FIFO into the OSR. There are two parameters used: one determines whether to block if the Tx FIFO is empty, and the other determines what to do if the OSR isn't empty enough as prescribed by a configurable parameter. The nonblocking pull moves the **X** scratch register into the OSR as a default value.

```
PULL block           ; Pull 32-bits from the Tx FIFO to the
                     ; OSR blocking to wait for data
PULL noblock         ; Pull from Tx FIFO if there is data
                     ; else copy X into the OSR
```



```
PULL ifempty block      ; Blocking pull, but only if OSR
                        ; is sufficiently empty
PULL ifempty noblock    ; Nonblocking pull, but only if
                        ; OSR is empty
```

Note There is an autopull configuration that is often used to do this automatically, saving an instruction.

MOV

MOV moves data from the source to the destination, with an option to either reverse the bits or perform a one's complement. The sources are

- PINS
- X
- Y
- NULL
- STATUS
- ISR
- OSR

The destinations are

- PINS
- X
- Y
- EXEC
- PC

- ISR
- OSR

Use ! or ~ for one's complement and :: to reverse the bits. Some examples are

```
MOV X, ~Y          ; Move the one's complement of Y to X
MOV X, ::Y         ; Move Y to X, reversing all the bits
MOV X, STATUS      ; Move the configured status to X
MOV EXEC, X        ; Execute the contents of X as an instruction
MOV PC, Y          ; Jump to the instruction specified by Y
```

The **STATUS** value can be configured to serve a few purposes, like indicating whether a FIFO is full or empty.

IRQ

IRQ sets or clears an interrupt either to the ARM CPU or to another PIO.

- Interrupts 0–3 are routed to the ARM CPU.
- Interrupts 4–7 are routed to the appropriate PIO in the same bank.

We'll talk about interrupts in Chapter 11, but for now, here are some examples:

```
IRQ SET 2          ; set interrupt 2,
                   ; won't wait for interrupt to be handled
IRQ CLEAR 2        ; clear interrupt 2
IRQ WAIT 2         ; set interrupt 2 and
                   ; wait for interrupt handler to clear it
IRQ SET 2 REL      ; interrupt number will be adjusted
                   ; by adding PIO number
```


SET

SET sets an immediate value to a destination. The immediate value is limited to five bits. The destinations are **PINS**, **X**, **Y**, and **PINDIRS**.

```
SET PINS, 1        ; Turn on the pins for this PIO
SET PINDIRS, 0     ; Turn the pins into input pins
SET X, 31          ; Set X to the value 31
```

About Controlling Timing

The program to flash the LEDs generated three square waves, one for each LED, with the one part offset differently for each LED. Most computer communications use square waves to represent binary data, the difference being that they operate at higher speeds than this flashing LEDs program. The hard part of implementing these protocols usually comes down to meeting the precise timing requirements in the electronics specs. The PIO processor has several features that help provide precise timing for these over the wire communication protocols. First, we'll look at how to control the speed our program executes at.

About the Clock Divider

By default, each PIO instruction executes in one system clock cycle, unless it has some sort of wait on an external event. The system clock runs at 125MHz, and the PIO will execute each instruction at this speed. For most protocols, this is too fast, and techniques to slow down are required like delaying loops. The PIO has a configuration to slow down how fast it operates via a clock divider. Based on a couple of registers, a number is divided into the system clock, and the PIO will operate at that speed. The valid values for the clock divider run from 1 to 65536 in increments of 1/256. The easiest way to configure this is via the RP2040 SDK function:

```
static inline void sm_config_set_clkdiv(
          pio_sm_config *c, float div);
```

where you pass the clock divider in as a floating-point number and the SDK splits it apart to set the integer and fractional clock divider hardware registers correctly.

To use the clock divider in our flashing LEDs program, we need to configure the clock divider in the **blink_program_init** function from **blink.pio** as shown in Listing 10-4.

Listing 10-4. The blink_program_init Function Setting the Clock Divider.

```
void blink_program_init(PIO pio, uint sm, uint offset,
         uint pin, float clkdiv) {
   pio_gpio_init(pio, pin);
   pio_sm_set_consecutive_pindirs(pio, sm, pin, 1, true);
   pio_sm_config c = blink_program_get_default_config(offset);
   sm_config_set_clkdiv(&c, clkdiv);
   sm_config_set_set_pins(&c, pin, 1);
   pio_sm_init(pio, sm, offset, &c);
}
```

Then we need to call it with

```
blink_program_init(pio, sm, offset, pin, 65536.0f);
```

Next, adjust our delay loops with

```
pio->txf[sm] = clock_get_hz(clk_sys) / freq / 65536;
```


Since the desired frequency is 5Hz, we reduced the delaying loop from 125,000,000/5 = 25,000,000 to 125,000,000/5/65,536 = 381.

The clock divider affects the speed of everything running on the PIO; however, we also have fine control of how long each individual instruction executes.

About the Delay Operand

Each PIO instruction has five bits set aside for delay and side setting. Side-set will be discussed shortly; in the meantime, we use all five bits for delay. The delay is specified in square brackets after the instruction and with all five bits has values of 0 to 31, for example:

```
MOV X, Y [31]
```

The **MOV** instruction is executed in one cycle and then waits 31 cycles before proceeding, making the instruction take 32 cycles in total.

When this is incorporated into the flashing LEDs program, the delay loops are eliminated entirely, as long as the LEDs flash at 10Hz rather than 5Hz. This is easily discernible to us poor slow humans. We could go a little slower, but this gives a good example of using instruction delay slowing the program down. This is combined with using the clock divider as well. The PIO Assembly code is shown in Listing 10-5.

Listing 10-5. PIO Code to Flash the LEDs Without a Delay Loop

```
.program blink
.wrap_target
    set pins, 1 [31]  ; Turn LED on
    mov x, x [31]
    mov x, x [31]
    mov x, x [31]
    mov x, x [31]
```

```
    mov x, x [31]
    set pins, 0 [31]  ; Turn LED off
    mov x, x [31]
    mov x, x [31]
    mov x, x [31]
    mov x, x [31]
    mov x, x [31]
    set pins, 0 [31]  ; Turn LED off
    mov x, x [31]
    mov x, x [31]
    mov x, x [31]
    mov x, x [31]
    mov x, x [31]
.wrap                 ; Blink forever!
```

Note We could also use the **NOP** instruction alias:

```
NOP [31]
```

This is an assembler alias to **MOV X,X** for readability.

Each section has six instructions:

- One to set the pin
- Five no-operations

To use up, 6 × 32 = 192 clock cycles.

This is a waste of the small 32-instruction PIO memory, but it demonstrates a timing control technique. Change the SLEEP_TIME as

```
#define SLEEP_TIME 100
```

Adjust the clock divider to

```
blink_program_init(pio, sm, offset, pin, 65104.17f);
```


See Exercise 10-1 for why we changed to this value. Slowing the RP2040 PIOs to something human readable is only barely possible; however, at computer to computer speeds, the techniques in this section are extremely powerful. Next, we see how to control the pins without using **SET** instructions.

About Side-Set

Side-set lets each instruction set up to five pins while executing. This is useful for controlling separate control pins or attaining maximum speed by eliminating **SET** instructions. Side-set uses the same bits as delay, so configuring bits for side-set reduces the number of bits available for delay, reducing the maximum delay time. By default, when side-set is configured, every instruction in the program will do a side-set, but you can configure the PIO to make side-set optional. The downside is that this uses one bit of the five bits available to specify side-set or delay. Listing 10-6 contains the PIO Assembly Language to use side-set.

Listing 10-6. PIO Program to Flash the LEDs Using Side-Set

```
.program blink
.side_set 1
.wrap_target
    mov x, x side 1 [15]  ; Turn LED on
    nop side 1 [15]
    mov x, x side 1 [15]
    mov x, x side 1 [15]
    mov x, x side 1 [15]
    mov x, x side 1 [15]
    mov x, x side 0 [15]  ; Turn LED off
    mov x, x side 0 [15]
    mov x, x side 0 [15]
```

```
    mov x, x side 0 [15]
    mov x, x side 0 [15]
    mov x, x side 0 [15]
    mov x, x side 0 [15]  ; Turn LED off
    mov x, x side 0 [15]
    mov x, x side 0 [15]
    mov x, x side 0 [15]
    mov x, x side 0 [15]
    mov x, x side 0 [15]
.wrap                     ; Blink forever!
```

This program flashes twice as fast, since we use one of the delay bits for side-set. Therefore, the delays are reduced from 31 to 15. The program is a collection of **NOP** instructions, where all the work is done by side-set, delay, and configuration.

The **.side_set** assembler directive tells the assembler how many side-set bits to configure and whether they are optional or not. This is necessary for the assembler to provide meaningful error messages and generate code correctly.

In the **blink_program_init routine**, change the **sm_config_set_set_pins** function to

```
sm_config_set_sideset_pins(&c, pin);
```

Since it's running twice as fast, change the definition of **SLEEP_TIME** to 50.

Programming the PIOs is a combination of code and configuration, we conclude with remaining configuration options.

More Configurable Options

This is a quick list of configuration options to be aware of, all of which can be set via RP2040 SDK functions:

1. Many PIO data functions only send or receive data; hence, they only use one of the RX or TX FIFOs. By default, each FIFO is four words, but you can configure one FIFO to be eight words, making the other 0.

2. You can often eliminate **PUSH** and **PULL** instructions by configuring **autopush** or **autopull.** These options will cause the **PUSH** and/or **PULL** to happen when a configured data threshold is reached.

3. Each PIO learned so far only writes to one GPIO pin. However, it has a 32-bit output register for writing to the pins, so all the pins are written to at once. This is why the various instructions that read or write the pins can process more than one bit.

4. Interpreting data as an instruction has not yet been presented, but the **MOV EXEC** and **OUT EXEC** functions can do this, allowing interesting ARM to PIO communication techniques and circumventing the 32-instruction limit.

5. There are many PIO examples in the **pico-examples github**. The best way to create a new PIO program is to find something similar in the examples and then modify it for the differences.

Summary

This was a whirlwind introduction to programming the PIO coprocessors contained in the RP2040. These are powerful processors for offloading communication functions from the two ARM CPU cores. We introduced this PIO functionality and viewed an example program to flash the LEDs. Next, we looked at all the instructions in detail and then studied program timing by modifying the flashing LEDs program to use all the various techniques. Then we looked at side-set to control GPIO pins and reviewed other useful configuration items.

In Chapter 11, we look at how to catch interrupts from internal and external devices and how to set interrupts from software.

Exercises

10-1. The system clock is 125,000,000MHz; each group of instructions executes in 6 * 32 = 192 clock cycles. Calculate the system clock divider to get a flash rate of 10Hz or ten times per second.

10-2. Using **side-set**, how fast can you get a square wave's frequency to cycle?

10-3. Write a PIO program to change the pin direction as directed by the ARM CPU. This would be like the program in Chapter 9. The ARM still does a lot of work, but this is good practice at sending data or instructions from the ARM to a PIO.

10-4. In the first example program in this chapter, remove the **SET** instruction by placing side-set on the **JMP** instructions.

10-5. The **gdb debugger** doesn't know about the PIO processors, and there isn't a **printf** statement for the PIOs. What are some possible techniques to debug a PIO program? Think about sending values to the ARM CPU for printing.

CHAPTER 11

How to Set and Catch Interrupts

All the various iterations of the flashing LEDs program had one thing in common: they were one large loop using different methods to control the timing of the flashing. If this was part of a larger program that was doing other tasks, such as driving a robot, then putting in hooks everywhere to check if the LEDs need processing is annoying and can easily lead to bugs. Another approach is to set a timer interrupt; here, we program a timer, which, when it goes off, interrupts our program to process the LEDs. This way we don't need a loop, nor do we need to integrate the handling of the LEDs into other parts of a larger program. In this chapter, we look at interrupts on the RP2040, how they work, and how to put them to use.

In general, when handling I/O, often, data is received randomly, and we just need notification when it is there to process it. Interrupts provide a great way to do this. The ARM Cortex-M0+ has powerful interrupt support and is well worth looking at. Before getting into the details, here is an overview of the RP2040's interrupt mechanisms.

Overview of the RP2040's Interrupts

The ARM Cortex-M0+ supports 32 separate interrupt sources, and the RP2040 implements 26, leaving six unused. Each of these interrupt sources wires an interrupt source, whether an internal or external device, to the

S. Smith, *RP2040 Assembly Language Programming*,
https://doi.org/10.1007/978-1-4842-7753-9_11

Nested Vector Interrupt Controller (**NVIC**). The NVIC knows the priority of each interrupt and decides if it needs to interrupt the CPU. When it interrupts the CPU, it saves the state of the running program and jumps to an interrupt handler defined in the interrupt vector table (**IVT**) located within memory. When the interrupt handler finishes processing the interrupt, it returns, and the CPU restores the state of the running program, letting it continue executing. Figure 11-1 diagrams this process.

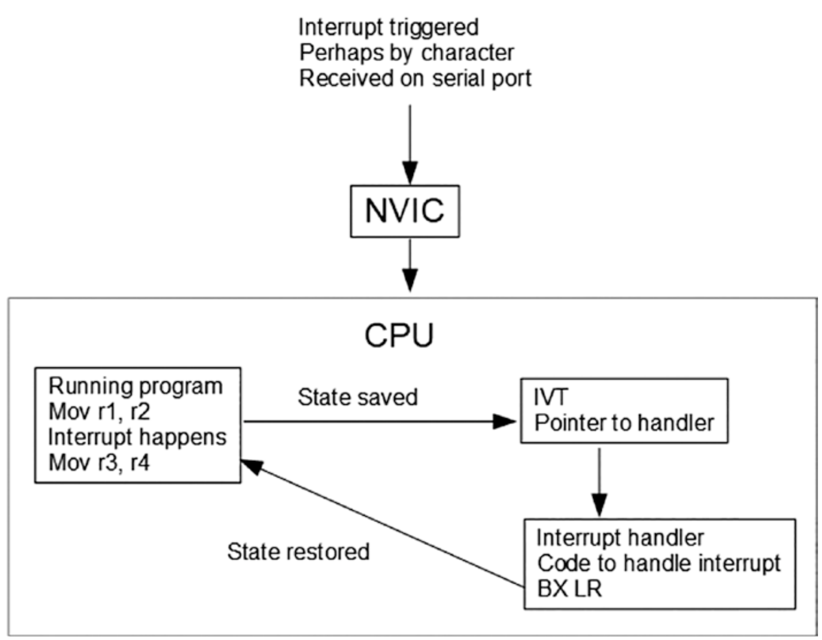

Figure 11-1. *Overview of the interrupt calling process*

With this overview in mind, let's dig into the various components in more detail starting with the list of interrupts.

About the RP2040's Interrupts

There are two sources of interrupts: those generated from within the CPU and those generated by devices external to the ARM CPU. Table 11-1 lists the ARM CPU internal interrupts.

Table 11-1. *The ARM's Internal Interrupts*

IRQ	Priority	Source	Comment
-1	0	Systick	ARM system 24-bit clock tick
-2	0	PendSV	Triggered by SVCall handler
-5	0	SVCall	Triggered by the SVC instruction
-13	-1	Hard fault	Triggered by nonrecoverable hardware failures
-14	-2	NMI	Nonmaskable interrupt
	-3	Reset	Triggered at power on or reset

Interrupts -3, -4, and -6 to -12 are unused and reserved for future use. The NMI interrupt is called when there is a fault in an interrupt handler routine, which is considered more serious than a fault happening in regular code. Table 11-2 lists the interrupts wired up to the ARM CPU inside the RP2040 SoC.

Table 11-2. *The RP2040's Interrupts and Their Priority*

IRQ	Priority	Source	Comment
0	2	Timer 0	Alarm 0
1	2	Timer 1	Alarm 1
2	2	Timer 2	Alarm 2
3	2	Timer 3	Alarm 3

(*continued*)

Table 11-2. (*continued*)

IRQ	Priority	Source	Comment
4	2	PWM	Interrupt when a slice is complete
5	2	USB	Data received
6	2	XIP	Off Chip ROM memory
7	2	PIO bank 0 - 0	
8	2	PIO bank 0 - 1	
9	2	PIO bank 1 - 0	
10	2	PIO bank 1 - 1	
11	2	DMA 0	Direct memory access
12	2	DMA 1	
13	2	GPIO	All the GPIO pins share this interrupt
14	2	QSPI	External flash memory
15	2	SIO 0	
16	2	SIO 1	
17	2	Clocks	
18	2	SPI 0	Data received, data sent, buffer overrun
19	2	SPI 1	
20	2	UART 0	11 possible reasons
21	2	UART 1	
22	2	ADC	FIFO reached threshold full
23	2	I2C 0	Data received or sent
24	2	I2C 1	
25	2	RTC	Real time clock

Let's look at how the RP2040 assigns an interrupt handler for each of these.

About the Interrupt Vector Table

When the RP2040 powers up, the IVT is located at address 0x00000000; however, the SDK's power-up routines move it to SRAM by setting a number of hardware registers associated with the ARM Cortex-M0+'s interrupt configuration. This table is a list of memory addresses, one for each interrupt. When an interrupt occurs, the ARM process jumps to the address stored for that interrupt.

The IVT contains an initial stack pointer (SP) to use after a reset interrupt or on power up and then the addresses of the handlers for the ARM internal interrupts, followed by the handlers for the connected devices.

Note For the ARM interrupts, the reserved interrupts still use a table spot, even though they aren't used.

Figure 11-2 shows the format of the IVT.

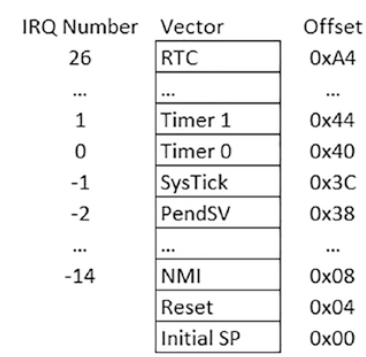

Figure 11-2. *Format of the interrupt vector table*

The easiest way to access the IVT is to read the hardware register where it is configured. PPB_BASE is the define for the memory address of the start of the ARM Cortex-M0+'s hardware registers; then **M0PLUS_VTOR_OFFSET** defined in **m0plus.h** is the offset to the IVT. The value of **M0PLUS_VTOR_OFFSET** is too large to fit in an immediate operand, so we need to load it from memory and then add these two numbers together to get the address of the IVT. The code snippet below shows this and loads the address of the IVT into **R1**.

```
#include "hardware/regs/addressmap.h"
#include "hardware/regs/m0plus.h"
...
      LDR R2, ppbbase
      LDR R1, vtoroffset
      ADD R2, R1
      LDR R1, [R2]
...
ppbbase:    .word PPB_BASE
vtoroffset: .word M0PLUS_VTOR_OFFSET
```

Place the address of the interrupt handler into the correct offset within this table. When the RP2040 jumps to an interrupt handler, it must first save the state of the running program.

About Saving Processor State

The state information of the processor is stored to the stack in a stack frame, whose contents are shown in Figure 11-3.

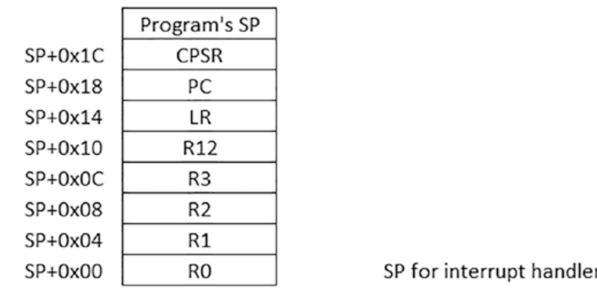

Figure 11-3. *Processor's saved state while interrupt handler runs*

In Chapter 7, the whole saving state was half in the called routine and half in the calling function. In this case of interrupts, the processor does the work for the calling routine. This stack frame is eight words in length and does not store registers **R4** to **R11**, so if they're needed, save and restore them in the handler routine. Since an interrupt can happen between any two instructions, the **CPSR** must be saved since the interrupt could happen between the instruction that sets the **CPSR** and then the instruction that acts on the **CPSR**. The overhead or minimum time an interrupt handler can take is the time to save these eight words to the stack and then restore them. The time depends upon whether they are cached or not. This sets a hard limit on how fast the RP2040 processes external data via the interrupt mechanism. Interrupts have a priority, and a higher-priority interrupt interrupts a lower-priority interrupt handler's routine, creating another stack frame.

About Interrupt Priorities

Each interrupt has a priority. All the externally connected interrupts can have four possible priorities from 0, 1, 2, and 3. With interrupts, the lower the number, the higher their priority is, so 0 has a higher priority than 3. By default, all these interrupts are set to 2 but can be changed via one of the ARM hardware configuration registers.

The interrupts nest, where if a higher-priority interrupt occurs while a lower-priority interrupt handler executes, then the processor interrupts the handler, creates a new stack frame, executes the handler for the higher-priority interrupt, removes its stack frame, and continues executing the lower-priority handler.

The ARM Cortex-M0+ implements optimizations to reduce the creation of stack frames:

1. If a higher-priority interrupt arrives while the CPU is creating the stack frame, then the CPU finishes creating the stack frame and lets the higher-priority interrupt use it, since the setup is the same for both. The NVIC remembers the original interrupt and runs it when the higher-priority interrupt finishes.
2. If a lower- or same-priority interrupt occurs while another interrupt runs, the processor won't tear down and recreate a stack frame; it passes control immediately to the next handler when the current handler finishes; this optimization applies to case 1 as well.

That completes the theoretical part of this chapter; now we look at how this all fits together in a real application.

Flashing LEDs with Timer Interrupts

There are many techniques to flash three LEDs; now we do it using the RP2040's built-in timer via an interrupt. In this example, we program one of the four RP2040 alarms to interrupt our program every 200ms to switch to the next LED. We implement the timer interrupt handler as a state machine, which increments the state, turns on or off each LED based on the state, and then programs the next timer interrupt. Listing 11-1 is the pseudocode for our alarm interrupt handler.

Listing 11-1. Pseudocode for the Alarm Interrupt Handler

```
Clear the interrupt
state = state + 1
switch  (state)
      Case 1:
            Turn on led 1, turn off leds 2 & 3
      Case 2:
            Turn on LED 2, turn off LEDs 1 & 3
      Case else:
            Turn on LED 3, turn off LEDs 1 & 2
            Set state = 0
Set the timer to go off in another 200ms
```

The state variable is a global variable located in SRAM and initialized to zero by the program. This example uses Assembly Language routines to manipulate the SIO hardware registers directly. The only SDK functions used are to print a count in the program's main loop, showing how the main part of the program can be written without worrying about the LEDs, which are entirely controlled by the interrupt handler. Before presenting the entire program, a bit of detail on the RP2040's alarm timer follows.

About the RP2040 Alarm Timer

The alarm timer is a 64-bit number that is incremented every microsecond. It supports four alarms, each on a separate interrupt IRQ0 to IRQ3. An alarm is programmed by setting a hardware register with a 32-bit number, and when the lower-order 32 bits of the timer match, an interrupt is fired. Hence, in our code, we read the timer's count, add 200,000 (200ms in microseconds), and then set the alarm. The locations of the hardware registers are in **timer.h**, with the base address in **addressmap.h**. The following is the code to do this with the assumption that **R0** contains 200,000.

```
#include "hardware/regs/addressmap.h"
#include "hardware/regs/timer.h"
...
      LDR R2, timerbase
      LDR R1, [R2, #TIMER_TIMELR_OFFSET]
      ADD R1, R0 @ R0 = 200,000
      STR R1, [R2, #TIMER_ALARM0_OFFSET]
...
timerbase: .word TIMER_BASE
```

When we receive a timer interrupt, we must clear the interrupt to acknowledge it was received, with

```
LDR R2, timerbase
MOV R1, #1 @ for alarm 0
STR R1, [R2, #TIMER_INTR_OFFSET]
```

After the new timer value is set, it's enabled with

```
LDR R2, timerbase
MOV R1, #1 @ for alarm 0
STR R1, [R2, #TIMER_INTE_OFFSET]
```

Besides programming the timers, when the program is initialized, we need to set the interrupt handler and enable the timer IRQ with the NVIC.

Setting the Interrupt Handler and Enabling IRQ0

Previously, we learned how to get the location of the IVT, and in this program, we configure our interrupt handler into it. Assuming we have the location of the IVT in R2, then we set the interrupt handler with


```
.EQU alarm0_isr_offset, 0x40
MOV R2, #alarm0_isr_offset @ slot for alarm 0
ADD R2, R1                 @ add the offset to the IVT
LDR R0, =alarm_isr         @ load address of our handler
STR R0, [R2]               @ save our routine to the IVT
```

By default, most interrupts are disabled; after all, why execute all these interrupt handlers if no one is using them? At program startup, we enabled IRQ0 to the NVIC with

```
     MOV R0, #1     @ alarm 0 is IRQ0 (bit 0)
     LDR R2, ppbbase
     LDR R1, clearint
     ADD R1, R2
     STR R0, [R1]
     LDR R1, setint
     ADD R1, R2
     STR R0, [R1]
...
clearint: .word M0PLUS_NVIC_ICPR_OFFSET
setint: .word M0PLUS_NVIC_ISER_OFFSET
```

In this case, follow the SDK recommendation to clear the interrupt and then enable it.

The Complete Program

Listing 11-2 contains the complete source code for this program and should be put in a file called **timeint.S**.

Listing 11-2. Flashing the LED via Timer Interrupts

```
@
@ Assembler program to flash three LEDs connected to the
@ Raspberry Pi GPIO using timer interrupts to trigger the
@ next LED to flash.
@

#include "hardware/regs/addressmap.h"
#include "hardware/regs/sio.h"
#include "hardware/regs/timer.h"
#include "hardware/regs/io_bank0.h"
#include "hardware/regs/pads_bank0.h"
#include "hardware/regs/m0plus.h"

      .EQU LED_PIN1, 18
      .EQU LED_PIN2, 19
      .EQU LED_PIN3, 20

      .EQU alarm0_isr_offset, 0x40

.thumb_func                @ Needed since SDK uses BX to call us
.global main               @ Provide program starting address

      .align  4            @ necessary alignment
main:
      BL stdio_init_all    @ initialize uart or usb

@ Init each of the three pins and set them to output
      MOV R0, #LED_PIN1
      BL  gpioinit
      MOV R0, #LED_PIN2
      BL  gpioinit
      MOV R0, #LED_PIN3
      BL  gpioinit
```



```
      BL  set_alarm0_isr   @ set the interrupt handler
      LDR R0, alarmtime    @ load the time to sleep
      BL  set_alarm0       @ set the first alarm

      MOV R7, #0           @ counter
loop:
      LDR R0, =printstr    @ string to print
      MOV R1, R7           @ counter
      BL  printf           @ print counter
      MOV R0, #1           @ add 1
      ADD R7, R0           @   to counter

      B   loop             @ loop forever

set_alarm0:
      @ Set's the next alarm on alarm 0
      @ R0 is the length of the alarm

      @ Enable timer 0 interrupt
      LDR R2, timerbase
      MOV R1, #1           @ for alarm 0
      STR R1, [R2, #TIMER_INTE_OFFSET]

      @ Set alarm
      LDR R1, [R2, #TIMER_TIMELR_OFFSET]
      ADD R1, R0
      STR R1, [R2, #TIMER_ALARM0_OFFSET]

      BX LR

.thumb_func                 @ necessary for interrupt handlers
@ Alarm 0 interrupt handler and state machine.
alarm_isr:
       PUSH {LR}            @ calls other routines
       @ Clear the interrupt
```

```
       LDR R2, timerbase
       MOV R1, #1           @ for alarm 0
       STR R1, [R2, #TIMER_INTR_OFFSET]

       @ Disable/enable LEDs based on state
       LDR R2, =state       @ load address of state
       LDR R3, [R2]         @ load value of state
       MOV R0, #1
       ADD R3, R0           @ increment state
       STR R3, [R2]         @ save state
step1: MOV R1, #1           @ case state == 1
       CMP R3, R1
       BNE step2            @ not == 1 check next
       MOV R0, #LED_PIN1
       BL  gpio_on
       MOV R0, #LED_PIN2
       BL  gpio_off
       MOV R0, #LED_PIN3
       BL  gpio_off
       B   finish
step2: MOV R1, #2           @ case state == 2
       CMP R3, R1
       BNE step3            @ not == 2 then case else
       MOV R0, #LED_PIN1
       BL  gpio_off
       MOV R0, #LED_PIN2
       BL  gpio_on
       MOV R0, #LED_PIN3
       BL  gpio_off
       B   finish
step3: MOV R0, #LED_PIN1    @ case else
       BL  gpio_off
```



```
       MOV R0, #LED_PIN2
       BL  gpio_off
       MOV R0, #LED_PIN3
       BL  gpio_on
       MOV R3, #0           @ set state back to zero
       LDR R2, =state       @ load address of state
       STR R3, [R2]         @ save state == 0

finish: LDR R0, alarmtime   @ sleep time
        BL  set_alarm0      @ set next alarm
        POP {PC}            @ return from interrupt

set_alarm0_isr:
       @ Set IRQ Handler to our routine
       LDR R2, ppbbase
       LDR R1, vtoroffset
       ADD R2, R1
       LDR R1, [R2]
       MOV R2, #alarm0_isr_offset @ slot for alarm 0
       ADD R2, R1
       LDR R0, =alarm_isr
       STR R0, [R2]

       @ Enable alarm 0 IRQ (clear then set)
       MOV R0, #1          @ alarm 0 is IRQ0
       LDR R2, ppbbase
       LDR R1, clearint
       ADD R1, R2
       STR R0, [R1]
       LDR R1, setint
       ADD R1, R2
       STR R0, [R1]

       BX  LR
```

```
@ Initialize the GPIO to SIO. r0 = pin to init.
gpioinit:
@ Initialize the GPIO
       MOV R3, #1
       LSL R3, R0          @ shift over to pin position
       LDR R2, gpiobase    @ address we want
       STR R3, [R2, #SIO_GPIO_OE_SET_OFFSET]
       STR R3, [R2, #SIO_GPIO_OUT_CLR_OFFSET]

@ Enable input and output for the pin
       LDR R2, padsbank0
       LSL R3, R0, #2      @ pin * 4 for register address
       ADD R2, R3          @ Actual set of registers for pin
       MOV R1, #PADS_BANK0_GPIO0_IE_BITS
       LDR R4, setoffset
       ORR R2, R4
       STR R1, [R2, #PADS_BANK0_GPIO0_OFFSET]

@ Set the function number to SIO.
       LSL R0, #3          @ each GPIO has 8 bytes of registers
       LDR R2, iobank0     @ address we want
       ADD R2, R0          @ add the offset for the pin number
       MOV R1, #IO_BANK0_GPIO3_CTRL_FUNCSEL_VALUE_SIO_3
       STR R1, [R2, #IO_BANK0_GPIO0_CTRL_OFFSET]
       BX  LR

@ Turn on a GPIO pin.
gpio_on:
       MOV R3, #1
       LSL R3, R0          @ shift over to pin position
       LDR R2, gpiobase    @ address we want
       STR R3, [R2, #SIO_GPIO_OUT_SET_OFFSET]
       BX  LR
```



```
@ Turn off a GPIO pin.
gpio_off:
       MOV R3, #1
       LSL R3, R0          @ shift over to pin position
       LDR R2, gpiobase    @ address we want
       STR R3, [R2, #SIO_GPIO_OUT_CLR_OFFSET]
       BX  LR

            .align  4           @ necessary alignment
gpiobase:   .word SIO_BASE      @ base of the GPIO registers
iobank0:    .word IO_BANK0_BASE @ base of io config registers
padsbank0:  .word PADS_BANK0_BASE
setoffset:  .word REG_ALIAS_SET_BITS
timerbase:  .word TIMER_BASE
ppbbase:    .word PPB_BASE
vtoroffset: .word M0PLUS_VTOR_OFFSET
clearint:   .word M0PLUS_NVIC_ICPR_OFFSET
setint:     .word M0PLUS_NVIC_ISER_OFFSET
alarmtime:  .word 200000
printstr:   .asciz "Couting %d\n"

.data
state:      .word 0
```

There is nothing special about the CMakeLists.txt file; it just needs to compile timeint.S. Notice that we did everything using just registers **R0** to **R3**, so we wouldn't need to save any registers ourselves.

That example used hardware interrupts; now let's view software interrupts.

About the SVCall Interrupt

The SVCall interrupt is a useful mechanism to implement operating system calls or to have the ability to call a routine without needing to link to it at compile time. This interrupt is triggered when a program executes the Supervisor Call (**SVC**) instruction:

```
SVC parameter
```

The parameter is an 8-bit immediate operand that allows 256 possible values. Linux uses this to call the operating system where the parameter is the Linux function number, and then the registers contain the parameters to that function where their exact values depend on which function it is.

Using the SDK

We haven't used the SDK yet, in order to provide a bare to the metal explanation of the interrupt process as is typically used by Assembly Language programmers. However, the SDK contains multiple useful functions for managing interrupts and for devices like the timer. It has support for higher-level functionality. It is worth reviewing what the SDK contains before implementing things yourself. Further, the complete source code for the SDK is posted to GitHub, which provides a wealth of sample code.

Summary

Interrupts are a mechanism where the running program can be interrupted at any point, and control is passed to a configured interrupt handler. Interrupts typically originate from hardware devices when new data arrives or needs attention. In this chapter, we studied the architecture of the ARM Cortex-M0+ interrupt system, set an interrupt handler, enabled

and configured interrupts, as well as learned how state is saved and how interrupts can be interrupted in a nested manner. We then looked in detail at the RP2040's timer device and how to use it to set alarms to interrupt our program on a regular basis. We then looked at a complete program to demonstrate all these concepts in action, again flashing the three LEDs. We then looked at software-triggered service interrupts and mentioned RP2040 SDK support.

We went quite far with the addition and subtraction of integers. We look at more mathematical operations in Chapter 12.

Exercises

11-1. Most software engineers work hard to make their interrupt handlers as fast as possible, leading many to be written in Assembly Language. Why do they do this? Does it matter how long an interrupt handler takes to execute?

11-2. If we debug the program, we see that the IVT is at the start of SRAM at memory location 0x20000000; why don't we hard-code that in our program and save a couple of instructions?

11-3. Modify the state machine in the sample program to create a pattern where two LEDs are lit at the same time.

11-4. Implement the sample program in C using the SDK.

11-5. Create a small Assembly Language program to use the SVC instruction and handle the interrupt, printing something so you know it was triggered.

CHAPTER 12

Multiplication, Division, and Floating Point

In this chapter, we return to using mathematics. We've already covered addition, subtraction, and a collection of bit operations on our 32-bit registers. Now we learn how to perform multiplication, division, interpolation, and floating point, starting with multiplication.

Multiplication

Integer 32-bit multiplication is built into the ARM Cortex-M0+, and the instruction set includes the **MUL** instruction:

```
MUL Rd, Rn
```

This instruction calculates **Rd** = **Rd** * **Rn** and executes in one clock cycle. Multiplying two 32-bit integers results in a 64-bit integer; however, this instruction simply discards or doesn't calculate the upper 32 bits. This works fine for smaller integers and equally well for signed or unsigned integers (Exercise 12-2), since the difference is in the discarded upper 32 bits.

S. Smith, *RP2040 Assembly Language Programming*,
https://doi.org/10.1007/978-1-4842-7753-9_12

A few examples:

```
MOV R2, #25
MOV R3, #5
MUL R2, R3     @ R2 = 125
NEG R3, R3     @ R3 = -5
MUL R2, R3     @ R2 = -625
```

Multiplication is straightforward within its limitations; now let's look at division.

Division

The ARM Cortex-M0+ doesn't have division instructions; however, the RP2040 adds a division coprocessor that performs a 32-bit integer division in eight clock cycles. This functionality is accessed through several hardware registers that we study. First, how do we determine when the division is complete? There is a division status register (**SIO_DIV_CSR**) with a ready bit that can be tested to determine if a calculation is complete. However, setting up a loop to test this bit is more work than it's worth (Exercise 12-3). The SDK recommends the macro in Listing 12-1 to wait eight cycles.

Listing 12-1. Macro to Delay Eight Cycles

```
.macro divider_delay
      // delay 8 cycles
      b   1f
1:    b   1f
1:    b   1f
1:    b   1f
1:
.endm
```

This delay macro takes advantage of the fact that a branch instruction clears the execution pipeline, so the next instruction needs to be reread. As a result, each branch instruction takes two cycles to execute, so executing four branch instructions is a sufficient delay. The ARM CPU doesn't detect this is really a **NOP** branching to the next instruction.

Rather than perform a delay loop, alternatively perform work that doesn't rely on the result of the division. For instance, if a calculation involves a division and other operations, first start the division, and perform the other operations while it executes. **Note**: This can be dangerous since you must ensure there are at least eight instructions in between. To perform division

1. Set the dividend and divisor registers; wait for the division to complete.
2. Read the remainder and quotient registers for the results.

There are two sets of dividend and divisor registers: one for signed integers and the other for unsigned integers. Listing 12-2 shows the code to perform a signed 32-bit integer division.

Listing 12-2. Example Division of Two Signed Integers

```
MOV R0, #11
MOV R1, #3
LDR R3, =SIO_BASE
STR R0, [R3, #SIO_DIV_SDIVIDEND_OFFSET]
STR R1, [R3, #SIO_DIV_SDIVISOR_OFFSET]
divider_delay
LDR R1, [R3, #SIO_DIV_REMAINDER_OFFSET]
LDR R0, [R3, #SIO_DIV_QUOTIENT_OFFSET]
```

Listing 12-3 shows a similar example for unsigned 32-bit integer division.

Listing 12-3. Example Division of Two Unsigned Integers

```
MOV R0, #100
MOV R1, #3
LDR R3, =SIO_BASE
STR R0, [R3, #SIO_DIV_UDIVIDEND_OFFSET]
STR R1, [R3, #SIO_DIV_UDIVISOR_OFFSET]
divider_delay
LDR R1, [R3, #SIO_DIV_REMAINDER_OFFSET]
LDR R0, [R3, #SIO_DIV_QUOTIENT_OFFSET]
```

When setting either the dividend or divisor register, any calculation in progress is cancelled and a new calculation started. If there are multiple calculations where one of these remains unchanged, they don't need to be set each time, and the calculation starts when the other is set.

Division is more work than multiplication but definitely easier and faster than creating a subtraction loop. But take care when performing divisions inside an interrupt handler.

About Division and Interrupts

In Chapter 11, the CPU did a good job of saving the CPU state before passing control to the interrupt handler. However, it provides no help with saving the state of the division coprocessor. If division isn't used in the interrupt handler, then there is no problem, as the division keeps calculating and is ready when the interrupted program continues. If division is performed in an interrupt handler, then preserve the values calculated for the interrupted program.

The **SIO_DIV_CSR** register contains a dirty bit to indicate a division was started, but the results haven't been retrieved yet. This is set when a calculation starts and cleared when the quotient is read. The remainder and quotient registers are both readable and writable. We read the values, do the work, and then write the original values back. Saving the stack frame and testing the dirty bit takes more than eight cycles, so any division is completed. This leads to an algorithm to preserve the division over an interrupt.

1. Test the dirty bit in the **SIO_DIV_CSR** register. If it is one, then read, and save the quotient and remainder registers.
2. Do the body of the interrupt handler.
3. If the quotient and remainder are saved, then restore them by writing them back to their registers.

Note This is only necessary if your interrupt handler does division; however, remember, if programmed in C, then the **/** and **%** operators use the division coprocessor. If you call an SDK routine, it might also perform a division. Not saving these values when needed can lead to some extremely hard-to-find bugs.

If writing a preemptive multitasker for the RP2040, then add these to the task state saved.

Division isn't the only mathematical coprocessor in the RP2040; there is also a hardware interpolator to look at next.

Interpolation

The RP2040 has two interpolator coprocessors for each ARM CPU core. These interpolators assist in several common algorithms used in audio and video processing. They can also assist in processing data being received into one of the RP2040's I/O devices. Consider the interpolators as a poor man's Digital Signal Processor (DSP). Many cell phone SoCs contain DSP processing blocks; however, at this point, Raspberry can't include a full DSP in their $4 chip.

DSPs typically perform full floating-point computations, contain instructions that are helpful for processing input signals, and have their own instruction sets. The RP2040's interpolators can assist with some of the same algorithms as full DSP chips but still rely on the ARM Cortex-M0+ to do much of the work. The interpolators contain their own registers and perform addition, multiplication, and some bit operations. They are intended to be used in loops where the result of each calculation cycle updates an accumulator. Each iteration step the interpolator performs takes one machine cycle.

The interpolator is complex and configurable. Rather than starting with the full picture, we'll build up piece by piece, starting with the simplest example of adding some integers.

Like division, the hardware registers for the interpolator are defined in **sio.h**; however, the offsets are too large to use as immediate mode offsets in **LDR** and **STR** instructions. This time, rather than perform the address calculations in the Assembly Language code, let the GNU assembler do the arithmetic, starting with a new base address:

```
INTERP_BASE: .word SIO_BASE + SIO_INTERP0_ACCUM0_OFFSET
```

where **SIO_INTERP0_ACCUM0_OFFSET** is the offset of the first interpolator register. Now the various registers can be accessed with instructions like

```
LDR R3, INTERP_BASE
STR R0, [R3, #(SIO_INTERP0_ACCUM0_OFFSET-SIO_INTERP0_ACCUM0_
OFFSET)]
```

We will use .**EQU** directives for each of these to keep the length of each line down. Let's look at the first and easiest example.

Adding an Array of Integers

To get used to working with the interpolator, first, is the simplest case of adding an array of 32-bit integers. Here, only one of the control registers and one of the two accumulators are accessed. Within the interpolator, there are two lanes, discussed later in this chapter; for this example, only lane 0 is used. Each lane has a control register that configures how the data flows and which operations to perform. In this simple example, we configure the lane control register **SIO_INTERP0_CTRL_LANE0** for raw addition only, which leaves most other things within the interpolator turned off. The accumulator is initialized to zero. Then every time a value is set to the **SIO_INTERP0_ACCUM0_ADD** register, the value is added to accumulator zero. At the end, read the value from accumulator zero for the final result. Listing 12-4 shows the Assembly Language code to perform this.

Listing 12-4. Using One of the Interpolators to Add an Array of Integers

```
.EQU INTERP0_CTRL_LANE0_OFF, (SIO_INTERP0_CTRL_LANE0_OFFSET-
SIO_INTERP0_ACCUM0_OFFSET)
.EQU INTERP0_ACCUM0_OFF, (SIO_INTERP0_ACCUM0_OFFSET-SIO_
INTERP0_ACCUM0_OFFSET)
.EQU INTERP0_ACCUM0_ADD_OFF, (SIO_INTERP0_ACCUM0_ADD_OFFSET-
SIO_INTERP0_ACCUM0_OFFSET)

interp: MOV R0, #0  @ init value for accum0
        MOV R1, #4  @ increment for array of nums
        MOV R2, #1  @ decrement for counter
        LDR R3, INTERP_BASE
        MOV R4, #1
        LSL R4, #SIO_INTERP0_CTRL_LANE0_ADD_RAW_LSB
        STR R4, [R3, #INTERP0_CTRL_LANE0_OFF]
```



```
        STR R0, [R3, #INTERP0_ACCUM0_OFF]
        LDR R7, numsumdata
        LDR R6, =sumdata
nextnum:LDR R4, [R6]
        STR R4, [R3,#INTERP0_ACCUM0_ADD_OFF]
        ADD R6, R1
        SUB R7, R2
        BNE nextnum
        LDR R0, [R3, #INTERP0_ACCUM0_OFF]
```

This is a complicated way to add an array of integers, especially when the ARM CPU can do this itself. A lot of the code is to initialize the interpolator and then the overhead of the loop, which reads and processes the array of numbers. Now, here is the complete set of interpolator registers:

1. **BASE0, BASE1, BASE2**: The numbers in these registers are input to the process.
2. **ACCUM0, ACCUM1**: The two accumulator registers, although ACCUM1 is an input when multiplying. Bit operations can be applied to the accumulators as part of each cycle.
3. **RESULT0, RESULT1, RESULT2**: The result registers that contain the calculations for each step. These can be fed back into the accumulators as part of the step.

The calculations the interpolator carries out depend on several parameters in the control registers. A typical calculation looks like

```
RESULT0 = lower8bits(ACCUM0) + BASE0
RESULT1 = rightshift8bits(ACCUM1) + BASE1
RESULT2 = RESULT0 + RESULT1 + BASE2
```

Then **RESULT0** and **RESULT1** can be fed into the accumulators for another iteration. The two accumulator calculations are referred to as the two calculation lanes and are configured separately. The bit operations are to **AND** by a series of 1 bits, perform a right shift, and perform a sign extension. These are typically used to extract byte data from a 32-bit word containing 4 bytes, perhaps 4 bytes of grayscale data.

Next is how to interpolate between values and why this coprocessor is called an interpolator.

Interpolating Between Numbers

To perform interpolation, we configure lane 0, containing accumulator 0 for blend mode. In blend mode, the interpolator calculates

```
RESULT1 = BASE0 + ACCUM1 * (BASE1 - BASE0)
```

This formula uses elements from both lanes, dedicating more of the interpolator. The multiplier is the lower 8 bits of **ACCUM1** after bit operations, interpreted as a fraction out of 255. This means we are multiplying the difference of **BASE1** and **BASE0** by a number between 0 and 1. This is interpolation: if **ACCUM1** is 0, then **RESULT1** is **BASE0**; if **ACCUM1** is 255, then **RESULT1** is **BASE1;** and any other value of **ACCUM1** will be between **BASE0** and **BASE1** by the fractional amount. The Assembly Language code to perform this calculation is contained in Listing 12-5. This program also calculates the sum of these interpolations, since **ACCUM0** isn't used otherwise. If **BASE0** is zero, then this calculates

$$\text{Result} = a_1 * b_1 + a_2 * b_2 + \ldots + a_n * b_n$$

This is the calculation used when multiplying a matrix by a vector, or a matrix by a matrix. This is helpful in machine learning, the limitation being that a_i needs to be normalized between 0 and 1; then the multiplication isn't as accurate as a full floating-point calculation but is much faster.

Listing 12-5. Code to Interpolate Between Some Numbers and Keep the Sum of the Results

```
.EQU INTERP0_BASE0_OFF, (SIO_INTERP0_BASE0_OFFSET-SIO_INTERP0_
ACCUM0_OFFSET)
.EQU INTERP0_BASE1_OFF, (SIO_INTERP0_BASE1_OFFSET-SIO_INTERP0_
ACCUM0_OFFSET)
.EQU INTERP0_ACCUM1_OFF, (SIO_INTERP0_ACCUM1_OFFSET-SIO_
INTERP0_ACCUM0_OFFSET)
.EQU INTERP0_PEEK1_OFF, (SIO_INTERP0_PEEK_LANE1_OFFSET-SIO_
INTERP0_ACCUM0_OFFSET)
.EQU INTERP0_CTRL_LANE1_OFF, (SIO_INTERP0_CTRL_LANE1_OFFSET-
SIO_INTERP0_ACCUM0_OFFSET)

@ Simple interpolation
interp2: MOV R0, #0        @ init value for accum1
         MOV R1, #4        @ increment for array of nums
         MOV R2, #1        @ decrement for counter
         MOV R3, #63
         MOV R8, R3
         LDR R3, INTERP_BASE
         MOV R4, #1
         LSL R4, #SIO_INTERP0_CTRL_LANE0_BLEND_LSB
         MOV R5, #1
         LSL R5, #SIO_INTERP0_CTRL_LANE0_ADD_RAW_LSB
         ORR R4, R5
         STR R4, [R3, #INTERP0_CTRL_LANE0_OFF]
         MOV r4, #248      @ 0xf8
         LSL r4, r4, #7    @ becomes 0x7c00
         STR R4, [R3, #INTERP0_CTRL_LANE1_OFF]
         STR R0, [R3, #INTERP0_ACCUM0_OFF]
         LDR R7, numsumdata
         LDR R6, =sumdata
```

```
nextnum2: LDR R4, [R6]
          STR R4, [R3,#INTERP0_BASE0_OFF]
          ADD R6, R1
          LDR R4, [R6]
          STR R4, [R3,#INTERP0_BASE1_OFF]
          STR R0, [R3,#INTERP0_ACCUM1_OFF]
          ADD R0, R8
          LDR R4, [R3,#INTERP0_PEEK1_OFF]
          STR R4, [R3,#INTERP0_ACCUM0_ADD_OFF]
          ADD R6, R1
          SUB R7, R2
          BNE nextnum2
          @ Read the sum stored in accumulator 0
          LDR R0, [R3, #INTERP0_ACCUM0_OFF]
```

We configure lane zero for blend mode and raw add mode. We could have figured out the necessary bit pattern and done this in fewer instructions, but since this is initialization code, it was left separate for readability.

For lane zero, we needed to configure it to not mask any bits; the configuration is to allow bits 0 to bits 31 through, which is what we want in this case; see Exercise 12-5.

To read the result registers, you read either the **PEEK** or **POP** register. PEEK reads the result without doing anything else. POP reads the value and also moves the result registers to the accumulators, depending on how the control registers are configured.

As the program goes through the loop, it reads the results but doesn't do anything with them. The program runs under **gdb**, and the results are viewed by single-stepping through the program.

The interpolator has other tricks like clamping the result range and configuring the movement of data in the lanes. The *RP2040 Datasheet* has a complete reference of all the functionality, and the RP2040 SDK samples

have a good selection of algorithms making use of the interpolator. Next, we learn how to use floating-point numbers and arithmetic from our Assembly Language programs.

Floating Point

The RP2040 doesn't have floating-point hardware. There is no floating-point coprocessor, so all floating-point instructions are done using the integer arithmetic instructions we studied. The GNU C compiler comes with software floating-point libraries for processors without floating-point support; however, these libraries don't know about the extras contained in the RP2040, like the integer division coprocessor.

To help with this, Raspberry included a fast floating-point library on the boot ROM. This library knows all the features of the RP2040 and uses the division coprocessor. The source code for the boot ROM is located in the raspberrypi/pico-bootrom GitHub repository. Most of this code is highly optimized Assembly Language and interesting to browse. Beware that even with these optimizations, floating-point routines are much slower than their integer counterparts.

While the **ADD**, **SUB**, and **MUL** instructions take one cycle to complete, the corresponding 32-bit floating-point routines take on average 70 cycles to complete. If transcendental functions like sine or cosine are used, they can take 700 cycles to execute. When programming in C, then the SDK automatically replaces routines in the standard C library with the routines located in the boot ROM, but they can be accessed directly from our Assembly Language code.

Note The original boot ROM version "A" only contained 32-bit floating-point functions, but the next version "B" added 64-bit floating-point support.

First, look at how the floating-point routines are found in the boot ROM.

About the Structure of the Boot ROM

The boot ROM contains the initial IVT; however, after this is a directory of the other services it offers. Table 12-1 is the layout of the first 32 bytes of the boot ROM.

Table 12-1. *The Layout of the Start of the Boot ROM*

Address	Contents	Description
0x00000000	Initial SP	Start of initial IVT
0x00000004	32-bit pointer	Boot reset interrupt handler
0x00000008	32-bit pointer	NMI interrupt handler
0x0000000c	32-bit pointer	Hardware fault interrupt handler
0x00000010	'M', 'u', 0x01	Magic numbers for sanity checking
0x00000013	Byte	Bootrom version, currently 0x1 or 0x2
0x00000014	16-bit pointer	Pointer to the function lookup table
0x00000016	16-bit pointer	Pointer to the data lookup table
0x00000018	16-bit pointer	Pointer to helper function

After the initial IVT follows four bytes that are informational. The next two pointers are the key to all the service functions available on the boot ROM. The second pointer, called the data lookup table, is really a second table of more functions, including the floating-point routines of interest. The pointer to the helper function at location 0x18 is to provide a way to access the contents of these tables without needing to hard-code values, providing flexibility to the designers of the boot ROM as the functionality is added in future versions. The pointers only need to be 16 bits since the boot ROM is limited in size and starts at address 0x0.

The definition of the data table is shown in Listing 12-6 from the file **bootrom_rt0.S** from the boot ROM's GitHub repository.

Listing 12-6. The Definition of the Boot ROM's Data Table

```
.global data_table
data_table:
    .byte 'G', 'R'
    .hword software_git_revision
    .byte 'C', 'R'
    .hword copyright
    .byte 'S', 'F'
    .hword soft_float_table
    .byte 'S', 'D'
    .hword soft_double_table
    .byte 'F', 'Z'
    .hword soft_float_table_size
// expose library start and end to facilitate copying to RAM
    .byte 'F, 'S'
    .hword mufp_lib_start
    .byte 'F, 'E'
    .hword mufp_lib_end
// expose library start and end to facilitate copying to RAM
    .byte 'D, 'S'
    .hword mufp_lib_double_start
    .byte 'D, 'E'
    .hword mufp_lib_double_end
    .hword 0
```

This table contains copyright information, version information, tables of function pointers, as well as the start and end of the various libraries. The reason for the start and end of the libraries is so these tables and routines can be copied from the boot ROM to static RAM if extra performance is required. Each element is a two-letter code followed by a 16-bit address or number. We could hard-code the offset to the software float table, but it's better to use the provided helper routine 0x18. This routine takes a pointer to one of the two tables and the code, and then it loops through the table finding the entry for the matching code, returning the halfword quantity associated with that code.

1. The address of the helper function is loaded with the following code:

```
.EQU helperfn, 0x18
MOV R5, #helperfn    @ address of the helper function
LDR R5, [R5]         @ load the helper function start
```

2. Set up the parameters to the helper function, then call it with:

```
.EQU datatable, 0x16
MOV  R3, #datatable   @ Load data table offset
LDRH R0, [R3]         @ Address of the data table
LDRH R1, code         @ Load the code SF for software
                        float
BLX  R5               @ call the helper function
MOV  R5, R0           @ Keep the SF table in R5
...
code: .ascii "SF"
```

This gives the table function pointers to the floating-point routines. The header file **pico/bootrom/sf_table.h** contains definitions for the offset into this table of each routine. In the code, the pointer moved to **R5** and called the various routines with code like

```
LDR R4, [R5, #SF_TABLE_FADD] @ Address of add routine
BLX R4                       @ Call the _fadd routine
```


This gives all the elements needed to add floating-point arithmetic to programs.

Note It's worth checking out the function in the function table that provides fast bulk memory and bit counting/manipulation functions.

Sample Floating-Point Program

Listing 12-7 is a program to add two floating-point numbers, print the sum, then calculate the sum's square root, and print that as well.

Listing 12-7. Program to Add Two Numbers and Calculate the Square Root

```
@
@ Examples of the floating point routines.
@

#include "pico/bootrom/sf_table.h"

.thumb_func                @ Necessary because sdk uses BLX
.global main               @ Provide program starting address
                             to linker

      .EQU datatable, 0x16
      .EQU helperfn, 0x18

main: BL   stdio_init_all  @ initialize uart or usb
      MOV  R3, #datatable  @ Load data table
      LDRH R0, [R3]        @ Address of the data table
      LDRH R1, code        @ Load the code SF for software float
      MOV  R5, #helperfn   @ address of the helper function
      LDR  R5, [R5]        @ load the helper function start
```

```
      BLX  R5              @ call the helper function
      MOV  R5, R0          @ Keep the SF table in R5
      LDR  R4, [R5, #SF_TABLE_FADD] @ Address of add routine
      LDR  R0, number1     @ First number to add
      LDR  R1, number2     @ Second number to add
      BLX  R4              @ Call the _fadd routine
      MOV  R7, R0          @ To calculate the square root later
      LDR  R4, [R5, #SF_TABLE_FLOAT2DOUBLE]
      BLX  R4              @ Call the _ftod routine
      MOV  R3, R1          @ Move results to input for printf
      MOV  R2, R0          @ ...
done: LDR  R0, =sumstr
      BL   printf          @ print the sum
      MOV  R0, R7          @ Original sum (32-bit)
      LDR  R4, [R5, #SF_TABLE_FSQRT]
      BLX  R4              @ Perform square root
      LDR  R4, [R5, #SF_TABLE_FLOAT2DOUBLE]
      BLX  R4              @ Call the _ftod routine
      MOV  R3, R1
      MOV  R2, R0
      LDR  R0, =sqrootstr
      BL   printf

loop:
      B    loop

.align 4
number1: .float  12.345
number2: .float  23.232
result:  .float  35.577
double:  .double 35.577
code:    .ascii "SF"
```



```
.data
           .align  4         @ necessary alignment
sumstr:    .asciz "The sum is %f\n"
sqrootstr: .asciz "Square root = %f\n"
```

This code is fairly straightforward, except how we pass the floating-point numbers to the **printf** routine.

Some Notes on C and printf

Besides the calls to addition and square root, there are two calls to float2double to convert our 32-bit floating-point number to a 64-bit number.

Note To run this program as is, version 2 of the boot ROM is required. If version 2 is not available, remove the calls to **float2double** and **printf** and read the result in **gdb**. The result of the addition is defined in the program to compare the result to.

The reason is that for a C function that takes a variable number of arguments, all floats are promoted to doubles. If a float is passed, then **printf** prints garbage or generates a fault. There is no way to pass a float to **printf**; it only takes a 64-bit double-precision floating-point number.

Passing 64-bit quantities in Chapter 7 wasn't discussed, but to do so, use two 32-bit registers if they are available or place them on the stack. As a parameter, the 64-bit quantity can either go in **R0** and **R1** or into **R2** and **R3**. Beyond that, they go on the stack. Placing 64-bit quantities in **R1** and **R2** is not allowed, and why we don't use **R1** in calls to **printf**. A 64-bit quantity can be returned in registers **R0** and **R1**, which is in the code.

The double-precision floating-point routines added to the boot ROM for version 2 only allow various conversions to and from double format. It doesn't provide routines for arithmetic, besides an arctan routine. These were added to perform 32-bit floating-point arithmetic but then provide results to other packages, like **printf**, that only take 64-bit numbers.

Summary

In this chapter, we studied the multiply (**MUL**) instruction. Even though the ARM Cortex-M0+ doesn't have a division instruction, designers at Raspberry provided the RP2040 with a division coprocessor that can perform a 32-bit integer division in eight CPU cycles. The division coprocessor performed divisions, and it was discussed how to use it in an interrupt handler. Next, the RP2040's interpolator coprocessor and how to use it to interpolate as well as perform multiply and accumulate operations was covered. The interpolator also has some bit manipulation operations that combine to give limited DSP-like capabilities for input data processing.

The RP2040 doesn't have a floating-point unit, so all floating-point operations must be performed using integer arithmetic and bit manipulations. However, Raspberry provided routines in the boot ROM that are faster than those included with the GNU C compiler, which use the hardware divider and other special knowledge of the RP2040 to achieve better performance. We looked at the structure of the boot ROM and how to call the routines located there. We wrote a program to add two floating-point numbers, calculate the square root, and print the result after converting it to a double-precision floating-point number.

So far in this book, everything was done on one of the two ARM Cortex-M0+ CPU cores contained in the RP2040. In Chapter 13, how to use the second CPU core and coordinate the work between the two CPUs is explained.

Exercises

12-1. Create a small program using the multiplication example and single-step through it in the debugger to ensure you understand how it works.

12-2. Examine the bits of calculating -1 * 4 to see why it works, either interpreting these as unsigned or signed integers.

12-3. Write the division delay loop as a loop testing the **SIO_DIV_CSR_READY** bit in the **SIO_DIV_CSR** register and proceeding once it changes to 1. What is the smallest number of instructions you can do this in? Does the loop ever perform a second iteration?

12-4. Create a small program using the divisions' examples and single-step through the code in **gdb** to ensure it works as expected.

12-5. In the interpolation example, we set lane one to the value 0x7c00. Look up the definition of the bits for the lane control register in the *RP2040 Datasheet* and see how this allows all the bits through, no masking.

12-6. Study the code for the helper function that scans the function or data tables in the boot ROM. The routine is written in Assembly Language; it is named table_lookup in **bootrom_rt0.S** or disassemble the code in **gdb**.

12-7. The area of a circle is $\pi * r^2$. Write a small Assembly Language program that uses the boot ROM's floating-point routines to calculate the area of circles with radius 1, 1.4, and 3. Print out the results.

CHAPTER 13

Multiprocessing

The RP2040 contains two ARM Cortex-M0+ CPU cores; in this chapter, we look at how to run code on the second processor. The second processor is in a power-conserving sleep state by default; we'll see how to wake it up and assign it work to process. Raspberry added two helpful features to the RP2040 for working with both CPU cores:

1. There are two FIFOs: one for core 0 to send data to core 1 and the other for core 1 to send data to core 0.
2. There are 32 spinlocks that can be assigned to control access to shared resources such as common memory areas.

Both are used in the sample programs, as well as three new ARM Assembly Language instructions for putting a CPU to sleep and waking it up. We start with these new instructions.

About Saving Power

Previously, waiting was done by entering tight loops; even the SDK's **sleep_ms** routine doesn't really sleep but rather enters a tight loop. This is fine, except that the CPU uses power to do this; however, the ARM CPU has a good power-saving mode. This can be important to save battery life, when running off a battery, or to reduce the heat generated by the RP2040

S. Smith, *RP2040 Assembly Language Programming*,
https://doi.org/10.1007/978-1-4842-7753-9_13

chip. Since most applications don't use the second CPU, it is put in a low-power mode by the boot ROM and often remains that way. Here are new instructions to wake up or put to sleep the second CPU, but these can also be useful in other circumstances. The three new instructions are

1. **SEV**: Send event. Causes a wakeup event to be sent to both processors.
2. **WFE**: Wait for an event. Enter a low-power state until an event is signaled. Will also wake up for a higher-priority interrupt or debug event.
3. **WFI**: Wait for an interrupt. Enter a low-power state until an asynchronous interrupt is received.

Note These instructions are classified as hints to the processor, meaning the processor is free to ignore them if it wants. Generally, put **WFE** or **WFI** instructions in a loop since they may wake up prematurely or may not go to sleep immediately. This is to allow the CPU to finish up other operations, such as writing cache data to the main memory before going to sleep.

Next, the instructions for the CPU core-to-core FIFO communication channel follow.

About Interprocessor Mailboxes

The RP2040 provides two FIFOs for interprocessor communications, and each FIFO contains eight 32-bit words. One FIFO is written by core 0 and read by core 1, the other read by core 0 and written by core 1. The same hardware registers are used by both, and the correct FIFO is used based on which does the reading or writing. The FIFO hardware is part of the RP2040's SIO hardware module, and hence, the defines for it are in **sio.h.** A CPU sends a message to the other CPU's mailbox with

```
      LDR R1, siobase
      STR R0, [R1, #SIO_FIFO_WR_OFFSET]
...
siobase: .WORD SIO_BASE
```

To read a message, the following code is used:

```
LDR R1, siobase
LDR R0, [R1, #SIO_FIFO_RD_OFFSET]
```

The preceding code is fine as long as there is room in the FIFO in the write case and if there is data available to read in the read case. To determine these, there is a status register. The status register has bits to tell

1. Whether the FIFO contains data
2. Whether the FIFO is full
3. Whether the FIFO was read when empty
4. Whether the FIFO was written to when full

Cases 1 and 2 are the most often used; cases 3 and 4 probably indicate a program bug. A more complete FIFO pop routine is given in Listing 13-1.

Listing 13-1. Interprocessor FIFO Read Routine

```
fifo_pop:
@ If there is data in the fifo, then read it.
        LDR R1, siobase
        LDR R0, [R1, #SIO_FIFO_ST_OFFSET]
        MOV R2, #SIO_FIFO_ST_VLD_BITS
        AND R0, R2
```



```
        BNE gotone
        WFE @ No data so go back to sleep
        B   fifo_pop @ try again if woken
gotone: LDR R0, [R1, #SIO_FIFO_RD_OFFSET]
        BX  LR
```

This routine is blocking; if there is no data, then it puts the processor to sleep and waits for data. For this to work, the routine called by the other core must add the **SEV** routine after writing to the FIFO to wake this processor up. With these tools, we'll look at how to get code running on the core 1 CPU.

How to Run Code on the Second CPU

When the RP2040 is powered on, both CPU cores receive a RESET interrupt, and the initial interrupt vector table (IVT) located at memory address 0x0 has the routine **_start** set as the interrupt handler. The first thing **_start** does is determine which CPU it is running as using

```
    LDR R0, =SIO_BASE
    LDR R1, [R0, #SIO_CPUID_OFFSET]
    CMP R1, #0             @ are we core 0?
    BNE wait_for_vector    @ not 0, so much be core 1
```

The **wait_for_vector** routine configures the second CPU for deep sleep mode and then waits on the interprocessor mailbox FIFO for data to be sent from the first CPU. The data it is waiting for is shown in Table 13-1.

Table 13-1. Data Sent to the Second CPU to Start It

Sequence	Contents	Description
0	0	Magic number
1	0	Magic number
2	1	Magic number
3	IVT	Interrupt vector table (use one for core 0)
4	SP	Top of stack (stack grows down)
5	Routine	Thumb routine to run (address must be odd)

We provide the same IVT as core 0 in our code, but a completely different IVT could be built for the second core. Keep in mind that it only receives interrupts if the interrupt is enabled by code running on that core. A stack in the data segment is defined and passes the top of the stack into the SP parameter.

Note Remember that the stack grows downward.

The last parameter is the address of the routine to run; it must be defined as a thumb function; since this routine is run via a **BLX** instruction, the address must be odd. This gives enough information to write a sample program to use the second core for processing. The code for all this is located in the **bootrom_rt0.S** file from the RP2040 boot ROM GitHub repository.

A Multiprocessing Example

To take an array of numbers and for each number to compute both the factorial and Fibonacci number, this program is easily written by calling two routines in turn on the same CPU core. However, performance is important, and both these computations are independent of each other. In this case, the Fibonacci number is calculated on core 0 and the factorial on core 1. First, we review Fibonacci numbers and factorials.

About Fibonacci Numbers

The Fibonacci numbers form a sequence (F_n) where each number is the sum of the preceding two numbers starting with 0 and 1, that is:

$F_0 = 0, F_1 = 1$

And

$F_n = F_{n-1} + F_{n-2}$

The first few numbers are

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

Fibonacci numbers appear in nature quite often and are closely related to the golden ratio.

About Factorials

The factorial of a positive integer **n**, denoted **n!**, is the product of all the positive integers less than or equal to n. Thus:

n! = n x (n-1) x (n-2) x … x 3 x 2 x 1

Factorials grow quickly, so in 32 bits, we can only calculate the first few of these. The first few factorials are

1, 2, 6, 24, 120, 720, 5040, 40320, …

Factorials are common in probability and combinatorics. With these in hand, we can review the complete program.

The Complete Program

Listing 13-2 presents the complete listing, which should go in a file **multicore.S** and accompany a standard **CMakeLists.txt** file.

Listing 13-2. Multiprocessor Program to Calculate Fibonacci Numbers and Factorials

```
@
@ Example using the second core for processing.
@

#include "hardware/regs/addressmap.h"
#include "hardware/regs/m0plus.h"
#include "hardware/regs/sio.h"

.thumb_func                 @ Necessary because sdk uses BLX
.global main                @ Provide program starting address
                              to linker

main: BL stdio_init_all     @ initialize uart or usb

      BL launch_core1

      MOV R4, #0            @ i = 0
      LDR R5, numNumbers
      LDR R6, =numbers
forloop: CMP R4, R5
      BGE mainloop
      LDR R0, [R6]          @ next number
      BL  fifo_push
      LDR R0, [R6]
      BL  fibonacci
      MOV R2, R0
      LDR R1, [R6]
```



```
      LDR R0, =fibprintstr
      BL  printf
      ADD R4, #1            @ i = i + 1
      ADD R6, #4            @ next word in numbers

      B   forloop
mainloop:
      B   mainloop

.align 4
numNumbers:   .WORD 5
numbers:      .WORD 3, 5, 7, 10, 12
fibprintstr:  .ASCIZ "Core 0 n = %d fibonacci = %d\n"
factprintstr: .ASCIZ "Core 1 n = %d factorial = %d\n"

.thumb_func
core1entry:
      PUSH {LR}
infinite: BL   fifo_pop     @ read number to calculate
      MOV R4, R0            @ keep n for the printf
      BL  factorial         @ call factorial
      MOV R2, R0            @ set parameters for printf
      MOV R1, R4
      LDR R0, =factprintstr
      BL  printf
      B   infinite          @ repeat for next number
      POP {PC}              @ never called.

fifo_push:
@ Push data to the fifo, without waiting.
      LDR R1, siobase
      STR R0, [R1, #SIO_FIFO_WR_OFFSET]
      SEV                   @ Wake up the other core
      BX  LR
```

```
fifo_pop:
@ If there is data in the fifo, then read it.
      LDR R1, siobase
      LDR R0, [R1, #SIO_FIFO_ST_OFFSET]
      MOV R2, #SIO_FIFO_ST_VLD_BITS
      AND R0, R2
      BNE gotone
      WFE                  @ No data so go back to sleep
      B   fifo_pop @ try again if woken
gotone: LDR R0, [R1, #SIO_FIFO_RD_OFFSET]
        BX  LR

fifo_drain:
@ Read the fifo 8 times to ensure its empty then wake up
@ the other core.
      LDR R1, siobase
      LDR R0, [R1, #SIO_FIFO_RD_OFFSET]
      LDR R0, [R1, #SIO_FIFO_RD_OFFSET]
      LDR R0, [R1, #SIO_FIFO_RD_OFFSET]
      LDR R0, [R1, #SIO_FIFO_RD_OFFSET]
      LDR R0, [R1, #SIO_FIFO_RD_OFFSET]
      LDR R0, [R1, #SIO_FIFO_RD_OFFSET]
      LDR R0, [R1, #SIO_FIFO_RD_OFFSET]
      LDR R0, [R1, #SIO_FIFO_RD_OFFSET]
      SEV
      BX  LR

launch_core1:
@ To start core1, writes the magic sequence:
@ 0, 0, 1, ivt, stack, routine
@ to core1's FIFO.
      PUSH {LR}
      BL   fifo_drain      @ Clear anything left over
```



```
      MOV  R0, #0
      BL   fifo_push
      BL   fifo_pop
      MOV  R0, #0
      BL   fifo_push
      BL   fifo_pop
      MOV  R0, #1
      BL   fifo_push
      BL   fifo_pop
      LDR  R2, ppbbase
      LDR  R1, vtoroffset
      ADD  R2, R1
      LDR  R0, [R2]
      BL   fifo_push
      BL   fifo_pop
      LDR  R0, =stack1_end
      BL   fifo_push
      BL   fifo_pop
      LDR  R0, =core1entry
      BL   fifo_push
      BL   fifo_pop
      POP  {PC}

.align 4
siobase:    .WORD SIO_BASE
ppbbase:    .word PPB_BASE
vtoroffset: .word MOPLUS_VTOR_OFFSET

@ R0 = fibonacci - in R0 since this is what is returned
@ R1 = f0
@ R2 = f1
@ R3 = i
@ R4 = n
```

```
fibonacci:
      PUSH {R4}
      MOV R4, R0             @ Move n to R4
      MOV R1, #0             @ Initial f0
      MOV R2, #1             @ Initial f1
      MOV R3, #2             @ Initial i = 2
loop: CMP R3, R4
      BGT done
      ADD R0, R1, R2         @ fibonacci = f0 + f1
      MOV R1, R2             @ f0 = f1
      MOV R2, R0             @ f1 = fibonacci
      ADD R3, #1             @ i = i + 1
      B   loop
done: POP {R4}
      BX  LR                 @ result is in R0

@ R0 = factorial
@ R1 = i
@ R2 = n
factorial:
       MOV R2, R0            @ Move n to R2
       MOV R0, #1            @ Initial factorial
       MOV R1, #2            @ i = 2
loop2: CMP R1, R2
       BGT done2
       MUL R0, R1            @ factorial *= i
       ADD R1, #1            @ i = i + 1
       B   loop2
done2: BX  LR                @ result is in R0

.align 4
.data
stack1:     .FILL 0x800, 1, 0
stack1_end: .WORD 0
```


The routines that calculate Fibonacci numbers and factorials are straightforward, implementing a simple FOR loop to calculate the desired number. It is worth reviewing these to ensure you understand how these simple calculations are performed in Assembly Language.

These three routines handle the interprocessor FIFO mailbox:

1. **fifo_drain**: Read the FIFO eight times to ensure it is empty. The SDK warns that there could be left-over data in the FIFO, and if run in the debugger, observe there is one value left over that needs clearing. It also calls **SEV** in case either processor has more processing to do after this happens.

2. **fifo_push**: Writes one word to the FIFO. This routine isn't blocking and doesn't check if the FIFO is full. In this case, the protocol means there is only one word in the FIFO at a time. The routine then calls **SEV** to wake up the other processor to read the value. See Exercise 13-2 to implement blocking.

3. **fifo_pop**: Checks the status register to see if there is data available; if there isn't, it goes to sleep by issuing a **WFE** instruction and loops back. If there is data, then it reads the data and returns it to the caller.

The routine to start the second core is **launch_core1**. This routine first clears any data left over in the FIFO and then executes the launch protocol to start the code running there. This involves writing the data it requires to the FIFO, after each word waiting for the same data to be echoed back. Listing 13-2 doesn't verify the data returned is the same as that sent. Strictly speaking, it should verify the core 1 code has responded with what it sent, and if not, then start over; see Exercise 13-1. Once core 1 is running, it listens to the interprocessor mailbox FIFO for data to process.

The main routine starts core 1 going and then reads the array of numbers targeted for performing the calculations. It pushes the number to the FIFO for core 1 to calculate the factorial and then goes ahead and calculates the Fibonacci number.

Each core prints its result using a **printf** statement. This works because the RP2040 ensures that **printf** is multiprocessor safe. On some systems the characters would be jumbled together, but in this SDK, the printing of the whole string is atomic. See Exercise 13-3 for an alternative way to do this.

Next are instructions on how to prevent the two CPU cores from stepping on each other.

About Spinlocks

The routines presented so far are completely independent and don't share any data or resources. This usually isn't the case when using two processors; they normally need to access shared data and that access needs to be regulated, so the two processors don't interfere with each other. For instance, if both processors update a table in memory, we don't want one processor overwriting the work of the other. When this goes wrong, this leads to hard-to-replicate bugs that are difficult to find.

The RP2040 provides 32 spinlocks to regulate access to shared resources. A spinlock is a resource that you try to acquire, but if someone else has it, it fails and the program spins using a closed loop until it's acquired. Like everything else, spinlocks are controlled by a set of hardware registers defined in **sio.h**. Of the 32 spinlocks, the first 16 are reserved for exclusive use by the SDK, and then the other 16 are available for use by programmers. If using the SDK, request a spinlock, and one will be allocated. Since we aren't using the SDK, we use spinlock 24 which is one the SDK will assign for exclusive use. Each spinlock has a hardware register that controls it, and then there is a separate hardware

register that will show the status of all 32 spinlocks, which can be useful for debugging, since reading it doesn't change any spinlock's state. To acquire a spinlock, read its hardware register, and if it reads nonzero, then you have successfully acquired it; if the value read is zero, then you need to spin to wait to acquire it. Listing 13-3 shows the code to lock a spinlock.

Listing 13-3. Code to Lock a Spinlock

```
        LDR R1, spinbase
repeat: LDR R0, [R1]
@ if spinlock is non-zero then we got it, else try again.
        CMP R0, #0
        BEQ repeat          @ spin
...
spinbase: .WORD SIO_BASE + SIO_SPINLOCK24_OFFSET
```

To release a spinlock, any value is written to the spinlock's hardware register. Listing 13-4 shows the code to release a spinlock.

Listing 13-4. Code to Unlock a Spinlock

```
LDR R1, spinbase
STR R0, [R1]       @ value written doesn't matter
```

Now let's look at a complete program that makes use of spinlocks.

Regulating Access to a Memory Table

This example program uses both CPU cores to populate a table of the numbers 0 to 99 and their squares. It also puts the core number in each row to mark the row as done, so we can see which core filled in each row. If spinlocks weren't used, then the cores would overwrite each other's work. Even though we mark a row as used first, there is a window of opportunity where both cores read a row as available and then both write to it at once and the core writing second wins. Using spinlocks to protect memory tables is common in operating systems, like Linux that supports multiple cores. Listing 13-5 is the complete program listing which should be called **spinlock.S**; after running, it will print the table of squares to see what work was done and which core filled in each row.

Listing 13-5. Program to Update Table of Squares Using Both Cores

```
@
@ Example using the second core for processing.
@ Protecting a memory table with a spin lock.
@

#include "hardware/regs/addressmap.h"
#include "hardware/regs/m0plus.h"
#include "hardware/regs/sio.h"

.thumb_func                 @ Necessary because sdk uses BLX
.global main                @ Provide program starting
                              address to linker

      .EQU numEntries, 100
      .EQU coreOffset, 0
      .EQU numOffset, 4
      .EQU numSquaredOffset, 8
      .EQU sizeTabRow, 12
      .EQU emptyRow, 255

main: BL stdio_init_all      @ initialize uart or usb

      BL launch_core1

      BL coremain
```



```
@ ensure everything finishes
      MOV R0, #255
      BL  sleep_ms

@ print out the table
      MOV R4, #0                 @ i = 0
      LDR R5, =numEntries
      LDR R6, =table
printtab:
      LDR R0, =printstr
      LDR R1, [R6, #coreOffset]
      LDR R2, [R6, #numOffset]
      LDR R3, [R6, #numSquaredOffset]
      BL  printf
      ADD R4, #1                @ i = i + 1
      ADD R6, #sizeTabRow
      CMP R4, R5                @ i = numEntries?
      BLT printtab

mainloop:
      WFE                       @ lower power now that we are
                                  done
      B   mainloop

.align 4
printstr: .ASCIZ "Core %d n = %d n * n = %d\n"
.align 4

.thumb_func
coremain:
      PUSH {R4, R5, R6, R7, LR}
      MOV  R4, #0               @ i = 0
      LDR  R5, =numEntries
```

```
      LDR  R6, =table
      MOV  R7, #emptyRow
forloop:
      @ lock spinlock
      BL   lockSpinLock
      @ determine if current row is free
      LDRB R0, [R6]
      CMP  R0, R7
      BNE  next                 @ not free, continue
      @ update table with core number, i, i*i
      LDR  R2, =SIO_BASE
      LDR  R2, [R2, #SIO_CPUID_OFFSET]
      @ unlock spinlock after marking row for this core
      BL   unlockSpinLock
      @ update next two fields
      STR  R2, [R6, #coreOffset]
      STR  R4, [R6, #numOffset]
      MOV  R0, R4
      MUL  R0, R0
      STR  R0, [R6, #numSquaredOffset]
@ Perform extra work, otherwise core 1 stays ahead
@ of core 0 and allocates all the table slots.
      .REPT 10
      NOP
      .ENDR
@ spinlock already unlocked, so jump ahead
      B   cont
next:
      @ unlock spinlock in case table entry taken
      BL  unlockSpinLock
cont: ADD R4, #1                @ i = i + 1
```



```
      ADD R6, #sizeTabRow
      CMP R4, R5
      BLT forloop

      @ Only return if we are core 0.
      LDR R2, =SIO_BASE
      LDR R2, [R2, #SIO_CPUID_OFFSET]
      CMP R2, #0
      BEQ ret
sleep: WFE
      B   sleep

ret: POP {R4, R5, R6, R7, PC}

lockSpinLock:
        LDR R1, spinbase
repeat: LDR R0, [R1]
@ if spinlock is non-zero then we got it, else try again.
      CMP R0, #0
      BEQ repeat
      BX  LR

unlockSpinLock:
      LDR R1, spinbase
      @ value written doesn't matter
      STR R0, [R1]
      BX  LR

fifo_push:
@ Push data to the fifo, without waiting.
      LDR R1, siobase
      STR R0, [R1, #SIO_FIFO_WR_OFFSET]
      SEV @ Wake up the other core
      BX  LR
```

```
fifo_pop:
@ If there is data in the fifo, then read it.
      LDR R1, siobase
      LDR R0, [R1, #SIO_FIFO_ST_OFFSET]
      MOV R2, #SIO_FIFO_ST_VLD_BITS
      AND R0, R2
      BNE gotone
      WFE                   @ No data so go back to sleep
      B   fifo_pop @ try again if woken
gotone: LDR R0, [R1, #SIO_FIFO_RD_OFFSET]
      BX  LR

fifo_drain:
@ Read the fifo 8 times to ensure its empty then wake up
@ the other core.
      LDR R1, siobase
      LDR R0, [R1, #SIO_FIFO_RD_OFFSET]
      LDR R0, [R1, #SIO_FIFO_RD_OFFSET]
      LDR R0, [R1, #SIO_FIFO_RD_OFFSET]
      LDR R0, [R1, #SIO_FIFO_RD_OFFSET]
      LDR R0, [R1, #SIO_FIFO_RD_OFFSET]
      LDR R0, [R1, #SIO_FIFO_RD_OFFSET]
      LDR R0, [R1, #SIO_FIFO_RD_OFFSET]
      LDR R0, [R1, #SIO_FIFO_RD_OFFSET]
      SEV
      BX  LR

launch_core1:
@ To start core1, writes the magic sequence:
@ 0, 0, 1, ivt, stack, routine
@ to core1's FIFO.
      PUSH {LR}
```



```
      BL   fifo_drain       @ Clear anything left over
      MOV  R0, #0
      BL   fifo_push
      BL   fifo_pop
      MOV  R0, #0
      BL   fifo_push
      BL   fifo_pop
      MOV  R0, #1
      BL   fifo_push
      BL   fifo_pop
      LDR  R2, ppbbase
      LDR  R1, vtoroffset
      ADD  R2, R1
      LDR  R0, [R2]
      BL   fifo_push
      BL   fifo_pop
      LDR  R0, =stack1_end
      BL   fifo_push
      BL   fifo_pop
      LDR  R0, =coremain
      BL   fifo_push
      BL   fifo_pop
      POP  {PC}

.align 4
siobase:    .WORD SIO_BASE
ppbbase:    .WORD PPB_BASE
vtoroffset: .WORD M0PLUS_VTOR_OFFSET
@ Spinlock 24 is first one available for exlucive use.
spinbase:   .WORD SIO_BASE + SIO_SPINLOCK24_OFFSET
```

```
.align 4
.data
stack1:     .FILL 0x800, 1, 0
stack1_end: .WORD 0
table:      .FILL numEntries * sizeTabRow, 1, emptyRow
```

This example is contrived, in that each processor performs exactly the same thing, leading to weird timing occurrences. Notice that after writing the data to the table, ten **NOP** instructions are performed. If this step is left out, then core 1 keeps ahead of core 0 and writes all the entries in the table; see Exercise 13-4.

In the main program after starting core 1 and filling in it's share of table entries, perform a sleep to make sure core 1 is finished processing. In a more robust system, a more deterministic manner should be used to ensure core 1 is complete; see Exercise 13-5.

In this chapter, code was written directly to the hardware registers; however, there are RP2040 SDK functions that can be used as follows.

A Word on the SDK

The RP2040 SDK contains routines to start work on the second CPU core, as well as to use the interprocessor FIFOs and spinlocks. The SDK routines are more robust than presented here since they have error checking. Unless there are specific use cases not covered by the SDK, use the routines contained there. The routines presented here are to demystify how the RP2040 works and provide intuition-based instructions on a deeper knowledge of how the operations work.

Summary

In this chapter, we learned how to use the second CPU core contained on the RP2040. Also, three new Assembly Language instructions were mastered to help conserve power. How to send messages between the two CPU cores and how to start programs running on the second core was explained. Since both CPU cores access the same memory on the RP2040, how to use spinlocks to control shared access to avoid the CPUs overwriting each other's work was learned.

In Chapter 14, how to connect an RP2040 microcontroller to the World Wide Web is covered.

Exercises

13-1. Add error checking to **launch_core1**. Break out the sending and receiving data to a separate routine that will check that the returned data is the same as the sent data and, if not, will return a failure code, starting the process over.

13-2. The **fifo_push** routine doesn't check if the FIFO is full before writing its data. Use the FIFO status register to check if the FIFO is full, and if so, then wait until it has free space and enter a low-power state while waiting, like how **fifo_pop** waits for data to arrive.

13-3. Each processor prints out the result of its calculation using **printf**. However, a more normal approach is to have core 1 write its result to the FIFO and have core 0 read it and then use the result, in this case, just printing it. Change the program to work this way, so core 1 is purely a computation service that is called to calculate factorials.

13-4. Remove the ten NOP instructions after the table row is written. How does that affect the results? Explain what is going on. How few NOPs can maintain an even workload?

13-5. Change the program so that core 1 writes a value to the interprocessor FIFO when it finishes its work. Next, have the main program wait for this value rather than calling a sleep function.

13-6. Both programs in this chapter make use of FOR-type loops to iterate through tables or to count through integers. Single-step through several of these loops in **gdb** to ensure you understand how they work.

13-7. Make the timer interrupt version of the flashing lights program from Chapter 11 more efficient by inserting a **WFI** when it doesn't have anything else to do.

CHAPTER 14

How to Connect Pico to IoT

In this chapter, we learn how to create a complete realistic microcontroller project written entirely in Assembly Language. We use our RP2040 device to collect data and then provide it to a central server. Since this is a book on Assembly Language and not electronics, components built into the RP2040 are used, rather than requiring extra components beyond what we already worked with. The built-in temperature sensor will be used to collect data; then the program will communicate with a server using UART0, which we've been using for debugging. The assumption is that a Raspberry Pi is being used for debugging and development, so this will be used as the server, and a Python program will be written to poll the various devices connected to it for data. This gives the opportunity to build a slightly larger program that uses everything learned to show how to put it all together. The program is divided into separate modules that are presented one by one. First, the RP2040's analog-to-digital converter and the built-in temperature sensor are presented.

S. Smith, *RP2040 Assembly Language Programming*,
https://doi.org/10.1007/978-1-4842-7753-9_14

About the RP2040's Built-in Temperature Sensor

Many sensor devices have no digital logic and work in an analog fashion; for instance, many temperature sensors, such as the RP2040's built-in one, are thermistors, which are resistors whose resistance varies depending on the ambient temperature. By measuring the voltage drop across a thermistor, Ohm's law can be used to determine its resistance and then use a provided formula to convert resistance to temperature. The RP2040 contains an analog-to-digital Converter (**ADC**) that measures the voltage received at a pin and returns a 12-bit number proportional to the voltage range. The range of voltages for the temperature sensor is 0 to 3.3V, so to convert from the 12-bit number to voltage, multiply by $3.3/2^{12}$. The "RP2040 Datasheet" then gives a formula to convert this voltage into degrees Celsius. Doing it this way requires floating-point arithmetic, which is not preferred. Combine these two formulas (see Exercise 14-4) to derive a formula that can be evaluated easily using only integer arithmetic:

```
Temp = 437 - (100 * rawADC) / 215
```

We want to divide the **rawADC** by 2.15, but multiplying both the numerator and denominator by 100 is a good trick to let us use only integer arithmetic. This is performed in the **calcTempCelc** function that uses the **intDivide** function as explained later in the math module.

The **ADC** has a status and control register that we use to enable both the **ADC** and the temperature sensor; these are turned off by default to save power. The **ADC** connects to four GPIO pins numbered 0 to 3 as well as the temperature sensor on port 4. The **ADC** can either do a round-robin scan on all its ports or read one port. Since only the temperature sensor is used, the control register is set to use port 4. The initialization routine builds up all the bits for this, so it can write it in one operation.

Note The **ADC** hardware registers are not single cycle with separate clear and set functions; all the bits used must be set every time it's written to, or read the port, add the bits used, and then write the value back.

When operating on the **ADC**, it takes several CPU cycles to perform its operation. This is why after initializing the ADC, the status register must be waited for until the device finishes powering up and is ready for use. Similarly, when we ask it to take a temperature reading, the program waits until the **ADC** finishes the operation.

Listing 14-1 contains the routines for programming the **ADC** controller and reading the temperature. Place these routines in a file called **adctemp.S**.

Listing 14-1. Routines to Activate the **ADC** Controller and Read the Temperature

```
@
@ Module to interface to the RP2040 ADC controller
@ as well as the built-in analog temperature sensor.
@

#include "hardware/regs/addressmap.h"
#include "hardware/regs/adc.h"

.EQU TEMPADC, 4

.thumb_func
.global calcTempCelc, initTempSensor, readTemp

@ Function to convert raw ADC data to degrees celsius.
@ Calculates degrees = 437 - 100 * R0 / 215
@
```

```
@ Registers:
@ Input:     R0 - raw 12-bit ADC value
@ Output:    R0 - degrees celsius
@ Other:     R1 - values to multiply or divide
@
calcTempCelc:
      PUSH {LR}                       @ needed since calls
                                        intDivide
      MOV  R1, #100
      MUL  R0, R1                     @ R0 = R0 * 100
      MOV  R1, #215
      BL   intDivide                  @ R0 = R0 / 215
      LDR  R1, tempcalcoff
      SUB  R0, R1, R0                 @ R0 = 437 - R0
      POP  {PC}

@ Initialize the ADC and temperature sensor.
@ No input parameters or return values.
@ Registers used: R1, R2, R3
initTempSensor:
@ Turn on ADC and Temperature Sensor
@ We set the bits to enable the ADC, the temp sensor
@ and select ADC line 4 (tempadc). All these bits are
@ in the ADC status register.
      MOV R1, #TEMPADC
      LSL R1, #ADC_CS_AINSEL_LSB
      ADD R1, #(ADC_CS_TS_EN_BITS+ADC_CS_EN_BITS)
      LDR R2, adcbase
      STR R1, [R2, #ADC_CS_OFFSET]

@ It takes a few cycles for these to start up, so wait
@ for the status register to say it is ready.
```



```
notReady2: LDR R1, [R2, #ADC_CS_OFFSET]
      MOV R3, #1
      LSL R3, #ADC_CS_READY_LSB
      AND R1, R3
      BEQ notReady2                    @ not ready, branch
      BX  LR

@ Function to read the temperature raw value.
@ Inputs - none
@ Outputs:  R0 - the raw ADC temperature value
@ Function requests a reading from the status reguiter
@  then waits for it to complete, then reads and returns
@  the value.
readTemp:
      LDR R2, adcbase
      LDR R1, [R2, #ADC_CS_OFFSET]      @ load status register
      ADD R1, #ADC_CS_START_ONCE_BITS   @ add read value once
      STR R1, [R2, #ADC_CS_OFFSET]      @ write to do it
notReady: LDR R1, [R2, #ADC_CS_OFFSET]  @ wait for read
      MOV R3, #1
      LSL R3, #ADC_CS_READY_LSB        @ done yet?
      AND R1, R3
      BEQ notReady
      LDR R0, [R2, #ADC_RESULT_OFFSET] @ read result
      BX  LR                           @ return value

             .align 4
adcbase:     .word ADC_BASE            @ base for analog to
                                         digital
tempcalcoff: .word 437
```

In this chapter, we separate the various functions into separate source code modules to reflect upon how to construct a larger program in a real situation. Now there's a raw **ADC** temperature reading, but before processing it further, consider how to send it to the server.

About Home-Brewed Communication Protocol

In this simple setup, the RP2040 board is connected directly to a Raspberry Pi with short cables. The output from the UARTs in both devices is low power and not suitable for long cables. However, there are many driver chips and devices available that can boost this signal to standards, like RS-422 and RS-485 that support long cables made of a twisted pair of wires. These can be hundreds of feet long and support multiple devices attached like Christmas tree lights. The design of the server to microcontroller protocol assumes this sort of architecture. The server polls for each device in turn for its data. The microcontroller only sends data to the server in response to a poll. The server sends out a poll consisting of three characters:

1. **SOH**: A start of header (ASCII character 1)
2. **ADDR**: The address of the device polled, in this case, ASCII '1' and up
3. **ETX**: An end of text character (ASCII character 3)

The terminal answers with a data packet of the form

1. **SOH**: A start of header (ASCII character 1).
2. **ADDR**: The address of the device, in our case, ASCII '1' and up.
3. **STX**: A start of text (ASCII character 2).

4. **Message**: The message data consists of printable ASCII characters.
5. **ETX**: An end of text character (ASCII character 3).

This is a simple protocol with no error checking (see Exercise 14-5), which simply demonstrates the start of a more full-featured protocol. Each device connected to the twisted pair wire needs to be configured with its own unique address. In this case, this is a program constant, so it needs to be changed and the program recompiled in each case.

The server will be implemented as a Python program that runs on the Raspberry Pi.

About the Server Side of the Protocol

The server program is implemented in Python, as this is an easy and popular way to program a Raspberry Pi. The routine to decode a received packet is implemented as a state machine, where it changes state if the correct character is received and returns to waiting for an **SOH** character if it isn't. The program polls a range of addresses and has a one-second timeout received, so if nothing is received in one second, it assumes the terminal isn't there and goes on to the next one. The best way to understand how the program works is to single-step through the parsing of a received packet to see how and when the state changes. Listing 14-2 contains this Python program, which should be stored in a file called **serpolling.py** and run from the Thonny Python IDE.

Listing 14-2. The Python Server Program

```
import serial
import time
from enum import Enum
```

```
class protocolState(Enum):
    SOH = 1
    ADDR = 2
    STX = 3
    MSG = 4

def sendPollreadResp(addr):
    ser.write(bytearray([1, addr, 3]))
    state = protocolState.SOH
    msg = bytes()
    while 1:
        x = ser.read()
        if x == b'':
                return( bytearray([0]) )
        elif state == protocolState.SOH:
                if x[0] == 1:
                    state = protocolState.ADDR
        elif state == protocolState.ADDR:
                if x[0] == addr:
                    state = protocolState.STX
                else:
                    return( bytearray([0]) )
        elif state == protocolState.STX:
                if x[0] == 2:
                    state = protocolState.MSG
                else:
                    return( bytearray([0]) )
        elif state == protocolState.MSG:
                if x[0] == 3:
                    return msg
                else:
                    msg = msg + x
```



```
    return( bytearray([0]) )

ser = serial.Serial(
        port = '/dev/serial0',
        baudrate = 115200,
        timeout=1
        )

while 1:
    for addr in range(49, 53):
        msg = sendPollreadResp(addr)
        print( msg )
```

With this, we have the server polling, so we'll go back to the RP2040 microcontroller to see how to use the UART to receive the poll and respond to it.

About the RP2040's UART

The UART device on the RP2040 chip takes bytes and serializes them and then sends them out on the wire bit by bit, or it reads bit by bit and assembles the bits into bytes for the consuming program. The UART contains receive and transmit FIFOs to buffer a few characters. There are programs within the SDK samples to demonstrate how to perform this functionality using the PIO coprocessors, but here we'll use one of the two built-in UART controllers. Like all connected hardware, there is a bank of hardware registers for controlling these. There are two registers for setting the baud rate, the speed the bits are put on the wire, and then two control registers for setting all the other properties. To send and receive data, there is a data register; then there is a collection of status registers that show what is going on.

The UART controller commands several control pins usually used with modems, but the Raspberry Pi Pico doesn't have a way to connect any of these to external GPIO pins, so a lot of the UART controllers' functionality can be ignored. Listing 14-3 contains the initialization routine for the UART along with routines to send and receive bytes of data. Magic numbers are set to the baud rate registers. The calculation of these is contained in the "RP2040 Datasheet" and left to Exercise 14-8 for the general case.

In the line control register **UARTLCR_H**, we set

1. 8-bit mode by setting the two WLEN bits to 1.
2. The FEN bit which enables the FIFOs.
3. Parity is not enabled, so it stays off.

In the control register **UARTCR**, we set the bits to

1. Enable the receiver
2. Enable the transmitter
3. Enable the UART

When reading a byte, we use the flag register **UARTFR** to determine the following:

1. When reading, if the receive FIFO isn't empty, then there's a character.
2. When transmitting, if the transmit FIFO isn't full, then it's possible to transmit.

We busy-wait on these conditions in the routines in Listing 14-3 that goes in a file called **muart.S**.

Listing 14-3. The Module for Controlling Serial Communications

```
@
@ Routines to handle the UART
@

#include "hardware/regs/addressmap.h"
#include "hardware/regs/uart.h"
#include "hardware/regs/io_bank0.h"
#include "hardware/regs/pads_bank0.h"

.thumb_func
.global initUART, readUART, sendUART

@ Function to initialize UART0.
@ Sets 115200 baud, 8 bits, no parity.
@ Enables the devices and configures the gpio pins.
@ No inputs or outputs.
@ Registers used: R0, R1.
@
initUART:
      PUSH {LR}
      LDR  R1, uart0base
      @ Set baud rate to 115200
      @ See the RP2040 datasheet for the magic values 67 and 52
      MOV R0, #67
      STR R0, [R1, #UART_UARTIBRD_OFFSET]
      MOV R0, #52
      STR R0, [R1, #UART_UARTFBRD_OFFSET]
      @ Set 8 bits no parity
      MOV R0, #(UART_UARTLCR_H_WLEN_BITS+UART_UARTLCR_H_FEN_BITS)
      STR R0, [R1, #UART_UARTLCR_H_OFFSET]
```

```
      @ Enable receive and transmit
      MOV R0, #3         @ enable TX and RX in one shot
      LSL R0, #UART_UARTCR_TXE_LSB
      ADD R0, #UART_UARTCR_UARTEN_BITS
      STR R0, [R1, #UART_UARTCR_OFFSET]

      MOV R0, #0
      BL  gpioInit
      MOV R0, #1
      BL  gpioInit

      POP {PC}

@ Function to read a character from the UART.
@ Waits for a character (no timeout) then reads the character.
@ Inputs: none
@ Outputs: R0 - character read
@ Registers used: R0, R1, R2
readUART:
      LDR R1, uart0base                 @ UART hardware
                                          register bank
      @ Wait for a character - that receive fifo isn't empty
waitr: LDR R0, [R1, #UART_UARTFR_OFFSET] @ read flag register
      MOV R2, #UART_UARTFR_RXFE_BITS     @ bits for fifo empty
      AND R0, R2
      BNE waitr                         @ set means fifo empty
      @ Read the character
      LDR R0, [R1, #UART_UARTDR_OFFSET]  @ read the character
      BX  LR

@ Function to send a character from the UART.
@ Waits for room in the transmit fifo then sends the character.
@ Inputs:  R0 - character to send
```



```
@ Outputs: none
@ Registers used: R0, R1, R2, R3
sendUART:
      LDR R1, uart0base
      @ Wait for transmitter free
waitt: LDR R3, [R1, #UART_UARTFR_OFFSET] @ read flag register
      MOV R2, #UART_UARTFR_TXFF_BITS     @ tx fifo full bits
      AND R3, R2
      BNE waitt                         @ set means fifo full
      @ Write the character
      STR R0, [R1, #UART_UARTDR_OFFSET]  @ send the character
      BX  LR

@ Function to initialize the GPIO to UART function.
@ Inputs: R0 - pin number
@
gpioInit:
@ Enable input and output for the pin
      LDR R2, padsbank0
      LSL R3, R0, #2     @ pin * 4 for register address
      ADD R2, R3         @ Actual set of registers for pin
      MOV R1, #PADS_BANK0_GPIO0_IE_BITS
      LDR R4, setoffset
      ORR R2, R4
      STR R1, [R2, #PADS_BANK0_GPIO0_OFFSET]

@ Set the function number to UART.
      LSL R0, #3         @ each GPIO has 8 bytes of registers
      LDR R2, iobank0    @ address we want
      ADD R2, R0         @ add the offset for the pin number
      MOV R1, #IO_BANK0_GPIO0_CTRL_FUNCSEL_VALUE_UART0_TX
      STR R1, [R2, #IO_BANK0_GPIO0_CTRL_OFFSET]
      BX  LR
```

```
             .align 4
uart0base: .word UART0_BASE
gpiobase:  .word SIO_BASE      @ base of the GPIO registers
iobank0:   .word IO_BANK0_BASE @ base of io config registers
padsbank0: .word PADS_BANK0_BASE
setoffset: .word REG_ALIAS_SET_BITS
```

Now that we can receive and transmit characters over the serial connection, we need a couple of utility math routines.

Mastering Math Routines

Integer division is used in two places in this program:

1. As part of the formula for converting from raw ADC to °C
2. To convert integers to ASCII

Move the bit of code that accesses the division coprocessor into a separate function. This code is straightforward and covered in Chapter 12.

The second routine needed is to convert binary integers into ASCII strings. This is done backward, by getting the least significant digit first and the most significant last, and then reversing the digits at the end. This is done by repeatedly dividing by 10. The remainder is the next digit, and the quotient will be divided again until there are no more digits. At the beginning, note if the number is negative and remember that a negative sign is added at the end, and then negate the number to make it positive. The algorithm works for negative numbers, except for the part where a digit is converted to ASCII by adding the ASCII '0' character.

At the end, add the negative sign if needed, and then reverse the string to get it in a human-readable form. The routines for this and division are in Listing 14-4, which should go in a file called **mmath.S**.

Listing 14-4. Routines for Division and Converting Integers to ASCII

```
@
@ Some useful math support routines including:
@   1. Divide two integers using the coprocessor
@   2. Convert an integer to ascii (in decimal)
@

#include "hardware/regs/addressmap.h"
#include "hardware/regs/sio.h"

.thumb_func
.global intDivide, itoa

@ macro to delay 8 clock cycles,
@ the time it takes to divide
.macro divider_delay
      // delay 8 cycles
      b    1f
1:    b    1f
1:    b    1f
1:    b    1f
1:
.endm

@ Function to divide two 32-bit integers
@ Inputs:    R0 - Dividend
@            R1 - Divisor
@ Outputs:   R0 - Quotient
@            R1 - Remainder
@
```

```
intDivide:
      LDR R3, =SIO_BASE
      STR R0, [R3, #SIO_DIV_SDIVIDEND_OFFSET]
      STR R1, [R3, #SIO_DIV_SDIVISOR_OFFSET]
      divider_delay
      LDR R1, [R3, #SIO_DIV_REMAINDER_OFFSET]
      LDR R0, [R3, #SIO_DIV_QUOTIENT_OFFSET]
      BX  LR

@ Function to convert a 32 bit integer to ASCII
@ Inputs:  R0 - number to convert
@          R1 - pointer to buffer for ASCII string
@ Outputs: R1 - contains the string
@
@ R7 - flag whether number positive or negative.
@ R6 - original buffer (since we increment R1 as we go along).
@ R4 - holds R1 around function calls (since they overwrite it)
@ R2, R3 - temp variables for reversing buffer
@
@ Builds the buffer in reverse by dividing by 10, placing the
@ remainder in the buffer and repeating, then at the end adding
@ a minus sign if needed. Then reverses the buffer to get
@ the correct order
itoa:
      PUSH  {R4, R6, R7, LR}
      MOV   R6, R1            @ original buffer
      MOV   R7, #0            @ assume number is positive
      CMP   R0, #0            @ is number positive
      BPL   convertdigits
      MOV   R7, #1            @ number is negative
      NEG   R0, R0            @ make number positive
```



```
convertdigits:
      MOV  R4, R1              @ preserve R1
      MOV  R1, #10             @ get least sig digit
      BL   intDivide
      ADD  R1, #'0'            @ convert digit to ascii
      STRB R1, [R4]            @ store ascii digit in buffer
      MOV  R1, R4              @ restore R1
      ADD  R1, #1              @ increment R1 for next character
      CMP  R0, #0              @ are we done (no more digits)?
      BEQ  finish              @ yes, go to finish up
      B    convertdigits       @ no, loop to do next digit

finish:
      CMP  R7, #0              @ is the number negative?
      BEQ  plus
      MOV  R0, #'-'            @ yes, add neg sign
      STRB R0, [R1]            @ store neg
      ADD  R1, #1              @ next position for null
plus: MOV  R0, #0              @ null terminator
      STRB R0, [R1]            @ null terminate
      SUB R1, #1               @ move pointer before null

      @ reverse the buffer
      SUB R2, R1, R6           @ length of buffer
revloop: LDRB R0, [R1]         @ get chars to reverse
      LDRB R3, [R6]
      STRB R0, [R6]            @ store reversed
      STRB R3, [R1]
      SUB  R1, #1              @ decrement end
      ADD  R6, #1              @ increment start
      SUB  R2, #2              @ done two characters
      BPL  revloop             @ still chars to process
      POP  {R4, R6, R7, PC}
```

With this, the modules needed to perform the various individual functions required are complete Next, the main program that uses all the functions is examined.

Viewing the Main Program

The main program implements a simple state machine to wait for a valid poll from the server. When received, it builds and sends the response message. It reads the temperature sensor and formats an ASCII message of the form “Temp: 23”. The message sent conforms to the protocol and is interpreted on the server. With the various modules that are now available, the main program is fairly simple.

The state machine is a simplified Assembly Language version of the one presented in the Python program. It is easier because there is no message received from the server, just **SOH** Addr **ETX**. The complete program is presented in Listing 14-5 and should go in a file called **iot.S**.

Listing 14-5. The Main Driving Program

```
@
@ Assembly Language program to answer polls from
@ a server and respond with the current temperature.
@

@ States for the state machine
.EQU SOH_State, 1
.EQU ADDR_State, 2
.EQU ETX_State, 3
```



```
@ Special protocol characters
.EQU SOHChar, 1
.EQU STXChar, 2
.EQU ETXChar, 3
.EQU TermAddrChar, 49

.thumb_func
.global main                    @ Provide program starting
                                  address

main:
@ Init the devices
      BL initTempSensor
      BL initUART

loop:
@ Starting state is waiting for SOH
      MOV R7, #SOH_State        @ state

waitforpoll:
      BL readUART               @ read next char

      @ switch( state = R7 )
      CMP R7, #SOH_State        @ are we waiting for SOH?
      BNE AddrStateCheck        @ no, check address state
      CMP R0, #SOHChar          @ did we read an SOH?
      BNE waitforpoll           @ no read another character
      MOV R7, #ADDR_State       @ yes switch to address state
      B   waitforpoll           @ wait for next character
AddrStateCheck:
      CMP R7, #ADDR_State       @ are we waiting for an address?
      BNE EtxStateCheck         @ no, check ETX state
      CMP R0, #TermAddrChar     @ is it our address?
      BEQ gotaddr               @ yes, goto gotaddr
```

```
      MOV R7, #SOH_State         @ no, go back to SOH state
      B   waitforpoll            @ get next char
gotaddr: MOV R7, #ETX_State      @ got address, so goto ETX
                                   state
      B   waitforpoll            @ get next char

EtxStateCheck:
      CMP R0, #ETXChar           @ did we get an ETX char?
      BEQ gotetx                 @ yes, goto gotetx
      MOV R7, #SOH_State         @ no, go back to SOH state
      B   waitforpoll            @ get next char

gotetx:
@ received a poll, so send a response packet
      MOV R0, #SOHChar
      BL  sendUART               @ send SOH
      MOV R0, #TermAddrChar
      BL  sendUART               @ send Address
      MOV R0, #STXChar
      BL  sendUART               @ send STX

      BL  readTemp               @ read the temperature

      BL  calcTempCelc           @ convert to degrees C

      LDR R1, =tempStr           @ msg template
      ADD R1, #6                 @ after Temp:
      BL  itoa                   @ raw temp value is still in R0

      LDR R5, =tempStr

@ Copy the msg string pointed to by R5 out the UART
nextchar: LDRB R0, [R5]
      CMP R0, #0                 @ String is null terminated
      BEQ done                   @ Are we done (at null)?
```



```
      BL  sendUART               @ No, then send the character
      ADD R5, #1                 @ Next character
      B   nextchar

@ Message is sent, so just need to send ETX character
done:
      MOV R0, #ETXChar
      BL  sendUART

@ This poll is finished, go back and wait for another
        B   loop                 @ loop forever

.data
@ template for temperature message string
tempStr: .asciz  "Temp:           "
```

The CMakeLists.txt file for this project is presented in Listing 14-6.

Listing 14-6. CMakeLists.txt File for This Project

```
cmake_minimum_required(VERSION 3.13)

include(pico_sdk_import.cmake)
project(iot C CXX ASM)

set(CMAKE_C_STANDARD 11)
set(CMAKE_CXX_STANDARD 17)

pico_sdk_init()

include_directories(${CMAKE_SOURCE_DIR})

add_executable(iot
  iot.S adctemp.S mmath.S muart.S
)

pico_enable_stdio_uart(iot 0)
pico_enable_stdio_usb(iot 1)
```

```
pico_add_extra_outputs(iot)

target_link_libraries(iot pico_stdlib)
```

Note Since UART is used to communicate, the **printf** output is configured to the UART. This means the **printf** output can't be viewed while debugging.

Here, the UART was used, since this connection is already available to the Raspberry Pi; however, there are other options, such as wireless, with some cost-versus-convenience trade-offs.

About IoT, Wi-Fi, Bluetooth, and Serial Communications

Internet of things (IoT) often refers to connecting microcontrollers to the Internet directly. However, the Raspberry Pi Pico does not come with Wi-Fi or Ethernet built into it. You can purchase Wi-Fi modules and interface them to the Raspberry Pi Pico using one of the serial protocols, such as I2C, UART, or SPI. There are RP2040-based boards that have Wi-Fi and Bluetooth built into them, such as the Seeed Studio Wio RP2040. These bundle a standard radio module and connect it to one of the serial communication set of ports on the RP2040 chip, typically one of the I2C ports. To use these, either use a vendor's supplied SDK or write directly to the device's serial interface either using the RP2040 SDK or by writing to the hardware registers directly.

The advantage of the UART serial protocol used is that the microcontroller doesn't need to know the Wi-Fi password to connect, similarly if Bluetooth is used as a wireless alternative. If Wi-Fi is used, be careful as if the microcontroller is stolen, the Wi-Fi credentials can be extracted from the ROM.

Having all the microcontrollers wired or wirelessly connected to the server, instead of using the Internet, prevents a lot of security problems. When the server they are connected with accesses the Internet, all Internet access is handled by a computer with a secure full-featured operating system such as Linux.

All these solutions are possible, and it comes down to trade-offs of cost, ease of installation, convenience, and security requirements. Often, serial wired communications are simple, cheap, and secure and work in an electrically noisy environment, like a factory. However, running a wire to every microcontroller can be a problem for homeowners who don't want to redo their drywall and prefer everything to be handled by their home Wi-Fi.

Summary

This chapter used all the things learned so far to create a complete Assembly Language program to read data from a device and then communicate it to a server program for processing or logging. The program used the hardware registers directly and didn't call any RP2040 SDK functions. Although Assembly Language is typically used to code highly specialized functions, which either require high performance or need to utilize machine instructions that aren't available from high-level languages, it is worth noting that in the microcontroller world, it is practical to write the entire program in Assembly Language.

Having read this far, you should have a good idea of how to write Assembly Language code for the RP2040 chip. You know how to write basic programs, as well as how to interface to all the devices that are bundled inside the RP2040.

Now it's up to you to go forth and experiment. The only way to learn programming is by doing. Think up your own Assembly Language projects. The RP2040 is a flexible device that can interface to nearly anything including any sensor or device that can be connected to the Arduino and Raspberry Pi systems.

Exercises

14-1. Change the program to report in degrees Fahrenheit rather than degrees Celsius.

14-2. The function itoa isn't safe, as it could overrun the provided buffer. Change the routine to take the buffer size as a third parameter and to ensure it doesn't write past the end of the provided buffer.

14-3. The Python program keeps adding to the msg variable until an ETX character is received. Change the program to have a maximum message length, which, if exceeded, will change the state back to waiting for an SOH character. Why is this a good practice?

14-4. Combine the formula for converting raw **ADC** to voltage with the temperature formula in the "RP2040 Datasheet" to derive our temperature formula.

14-5. The simple protocol has no error checking. One technique is to add an XOR checksum to the message. Simple XOR all the bytes of the message together and include the checksum before the ETX character. Implement this for our protocol. How do you ensure the checksum isn't one of the three special protocol characters?

14-6. The simple protocol has no authentication; should a terminal need to supply authentication information? What are the pros and cons of adding this?

14-7. Typical temperatures are around room temperature or 20°C; two digits positive. Setup some test cases for the itoa function to ensure it works properly for negative temperatures. What is a good selection of test cases to ensure it is working properly?

14-8. In the **initUART** function, the baud rate is hard-coded to 115200. Change the routine to take the baud rate as a parameter and perform the calculations explained in the "RP2040 Datasheet" to configure the two baud rate registers correctly.

APPENDIX A

ASCII Character Set

Here is the ASCII Character Set. The characters from 0 to 127 are standard. The characters from 128 to 255 are not standard and depend on geographic region and computer manufacturer among other things. The values of these characters are specified by a code page and the ones presented here are taken from code page 437, which is the character set of the original IBM PC.

Dec	Hex	Char	Description
0	00	NUL	Null
1	01	SOH	Start of header
2	02	STX	Start of text
3	03	ETX	End of text
4	04	EOT	End of transmission
5	05	ENQ	Enquiry
6	06	ACK	Acknowledge
7	07	BEL	Bell
8	08	BS	Backspace
9	09	HT	Horizontal tab

(*continued*)

S. Smith, *RP2040 Assembly Language Programming*,
https://doi.org/10.1007/978-1-4842-7753-9

Dec	Hex	Char	Description
10	0A	LF	Line feed
11	0B	VT	Vertical tab
12	0C	FF	Form feed
13	0D	CR	Carriage return
14	0E	SO	Shift out
15	0F	SI	Shift in
16	10	DLE	Data link escape
17	11	DC1	Device control 1
18	12	DC2	Device control 2
19	13	DC3	Device control 3
20	14	DC4	Device control 4
21	15	NAK	Negative acknowledge
22	16	SYN	Synchronize
23	17	ETB	End of transmission block
24	18	CAN	Cancel
25	19	EM	End of medium
26	1A	SUB	Substitute
27	1B	ESC	Escape
28	1C	FS	File separator
29	1D	GS	Group separator
30	1E	RS	Record separator
31	1F	US	Unit separator
32	20	space	Space

(*continued*)

Dec	Hex	Char	Description
33	21	!	Exclamation mark
34	22	"	Double quote
35	23	#	Number
36	24	$	Dollar sign
37	25	%	Percent
38	26	&	Ampersand
39	27	'	Single quote
40	28	(	Left parenthesis
41	29	)	Right parenthesis
42	2A	*	Asterisk
43	2B	+	Plus
44	2C	,	Comma
45	2D	-	Minus
46	2E	.	Period
47	2F	/	Slash
48	30	0	Zero
49	31	1	One
50	32	2	Two
51	33	3	Three
52	34	4	Four
53	35	5	Five
54	36	6	Six
55	37	7	Seven

(continued)

Dec	Hex	Char	Description
56	38	8	Eight
57	39	9	Nine
58	3A	:	Colon
59	3B	;	Semicolon
60	3C	<	Less than
61	3D	=	Equality sign
62	3E	>	Greater than
63	3F	?	Question mark
64	40	@	At sign
65	41	A	Capital A
66	42	B	Capital B
67	43	C	Capital C
68	44	D	Capital D
69	45	E	Capital E
70	46	F	Capital F
71	47	G	Capital G
72	48	H	Capital H
73	49	I	Capital I
74	4A	J	Capital J
75	4B	K	Capital K
76	4C	L	Capital L
77	4D	M	Capital M
78	4E	N	Capital N

(continued)

Dec	Hex	Char	Description
79	4F	O	Capital O
80	50	P	Capital P
81	51	Q	Capital Q
82	52	R	Capital R
83	53	S	Capital S
84	54	T	Capital T
85	55	U	Capital U
86	56	V	Capital V
87	57	W	Capital W
88	58	X	Capital X
89	59	Y	Capital Y
90	5A	Z	Capital Z
91	5B	[	Left square bracket
92	5C	\	Backslash
93	5D	]	Right square bracket
94	5E	^	Caret/circumflex
95	5F	_	Underscore
96	60	`	Grave/accent
97	61	a	Small a
98	62	b	Small b
99	63	c	Small c
100	64	d	Small d
101	65	e	Small e

(continued)

Dec	Hex	Char	Description
102	66	f	Small f
103	67	g	Small g
104	68	h	Small h
105	69	i	Small i
106	6A	j	Small j
107	6B	k	Small k
108	6C	l	Small l
109	6D	m	Small m
110	6E	n	Small n
111	6F	o	Small o
112	70	p	Small p
113	71	q	Small q
114	72	r	Small r
115	73	s	Small s
116	74	t	Small t
117	75	u	Small u
118	76	v	Small v
119	77	w	Small w
120	78	x	Small x
121	79	y	Small y
122	7A	z	Small z
123	7B	{	Left curly bracket
124	7C	|	Vertical bar

(continued)

Dec	Hex	Char	Description
125	7D	}	Right curly bracket
126	7E	~	Tilde
127	7F	DEL	Delete
128	80	Ç	
129	81	ü	
130	82	é	
131	83	â	
132	84	ä	
133	85	à	
134	86	å	
135	87	ç	
136	88	ê	
137	89	ë	
138	8A	è	
139	8B	ï	
140	8C	î	
141	8D	ì	
142	8E	Ä	
143	8F	Å	
144	90	É	
145	91	æ	
146	92	Æ	
147	93	ô	

(continued)

Dec	Hex	Char	Description
148	94	ö	
149	95	ò	
150	96	û	
151	97	ù	
152	98	ÿ	
153	99	Ö	
154	9A	Ü	
155	9B	¢	
156	9C	£	
157	9D	¥	
158	9E	Pts	
159	9F	ƒ	
160	A0	á	
161	A1	í	
162	A2	ó	
163	A3	ú	
164	A4	ñ	
165	A5	Ñ	
166	A6	ª	
167	A7	º	
168	A8	¿	
169	A9	⌐	
170	AA	¬	

(continued)

Dec	Hex	Char	Description
171	AB	½	
172	AC	¼	
173	AD	¡	
174	AE	«	
175	AF	»	
176	B0	░	
177	B1	▒	
178	B2	▓	
179	B3	│	
180	B4	┤	
181	B5	╡	
182	B6	╢	
183	B7	╖	
184	B8	╕	
185	B9	╣	
186	BA	║	
187	BB	╗	
188	BC	╝	
189	BD	╜	
190	BE	╛	
191	BF	┐	
192	C0	└	
193	C1	┴	

(*continued*)

Dec	Hex	Char	Description
194	C2	┬	
195	C3	├	
196	C4	─	
197	C5	┼	
198	C6	╞	
199	C7	╟	
200	C8	╚	
201	C9	╔	
202	CA	╩	
203	CB	╦	
204	CC	╠	
205	CD	═	
206	CE	╬	
207	CF	╧	
208	D0	╨	
209	D1	╤	
210	D2	╥	
211	D3	╙	
212	D4	╘	
213	D5	╒	
214	D6	╓	
215	D7	╫	
216	D8	╪	

(*continued*)

Dec	Hex	Char	Description
217	D9	┘	
218	DA	┌	
219	DB	█	
220	DC	▄	
221	DD	▌	
222	DE	▐	
223	DF	▀	
224	E0	α	
225	E1	ß	
226	E2	Γ	
227	E3	π	
228	E4	Σ	
229	E5	σ	
230	E6	µ	
231	E7	τ	
232	E8	Φ	
233	E9	Θ	
234	EA	Ω	
235	EB	δ	
236	EC	∞	
237	ED	φ	
238	EE	ε	
239	EF	∩	

(continued)

APPENDIX A ASCII CHARACTER SET

Dec	Hex	Char	Description
240	F0	≡	
241	F1	±	
242	F2	≥	
243	F3	≤	
244	F4	⌠	
245	F5	⌡	
246	F6	÷	
247	F7	≈	
248	F8	°	
249	F9	•	
250	FA	·	
251	FB	√	
252	FC	ⁿ	
253	FD	²	
254	FE	■	
255	FF		

APPENDIX B

Assembler Directives

This appendix lists a useful selection of GNU Assembler directives. It includes all the directives used in this book and a few more that are commonly used.

Directive	Description
.align	Pads the location counter to a particular storage boundary.
.ascii	Defines memory for an ASCII string with no NULL terminator.
.asciz	Defines memory for an ASCII string and adds a NULL terminator.
.byte	Defines memory for bytes.
.data	Assembles following code to the end of the data subsection.
.double	Defines memory for double floating-point data.
.dword	Defines storage for 64-bit integers.
.else	Part of conditional assembly.
.elseif	Part of conditional assembly.
.endif	Part of conditional assembly.
.endm	End of a macro definition.
.endr	End of a repeat block.
.equ	Defines values for symbols.

(continued)

S. Smith, *RP2040 Assembly Language Programming*,
https://doi.org/10.1007/978-1-4842-7753-9

Directive	Description
.fill	Defines and fills some memory.
.float	Defines memory for single-precision floating-point data.
.global	Makes a symbol global, needed if referenced from other files.
.hword	Defines memory for 16-bit integers.
.if	Marks the beginning of code to be conditionally assembled.
.include	Merges a file into the current file.
.int	Defines storage for 32-bit integers.
.long	Defines storage for 32-bit integers (same as .int).
.macro	Defines a macro.
.octa	Defines storage for 64-bit integers.
.quad	Same as .octa.
.rept	Repeats a block of code multiple times.
.set	Sets the value of a symbol to an expression.
.short	Same as .hword.
.single	Same as .float.
.text	Generates following instructions into the code section.
.word	Same as .int.

APPENDIX C

Binary Formats

This appendix describes the basic characteristics of the data types we have been working with.

Integers

Table C-1 provides the basic integer data types we have used. Signed integers are represented in two's complement form.

Table C-1. *Size, Alignment, Range, and C Type for the Basic Integer Types*

Size	Type	Alignment in bytes	Range	C type
8	Signed	1	–128 to 127	signed char
8	Unsigned	1	0 to 255	char
16	Signed	2	–32,768 to 32,767	short
16	Unsigned	2	0 to 65,535	unsigned short
32	Signed	4	–2,147,483,648 to 2,147,483,647	int
32	Unsigned	4	0 to 4,294,967,295	unsigned int
64	Signed	8	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	long long
64	Unsigned	8	0 to 18,446,744,073,709,551,615	unsigned long long

S. Smith, *RP2040 Assembly Language Programming*,
https://doi.org/10.1007/978-1-4842-7753-9

Floating Point

The RP2040 floating-point routines use the IEEE-754 standard for representing floating-point numbers. All floating-point numbers are signed.

Table C-2. *Size, Positive Range, and C Type for Floating-Point Numbers*

Size	Range	C type
32	1.175494351e-38 to 3.40282347e+38	float
64	2.2250738585072014e-308 to 1.7976931348623157e+308	double

Addresses

All addresses or pointers are 32 bits.

Table C-3. *Size, Range, and C Type of a Pointer*

Size	Range	C type
32	0 to 4,294,967,295	void *

APPENDIX D

The ARM Instruction Set

This appendix lists the core ARM Cortex-M0+ 32-bit instruction set, with a brief description of each instruction.

Instruction	Description
ADC, ADD	Add with carry, add
ADR	Load program or register-relative address (short range)
AND	Logical AND
ASR	Arithmetic shift right
B	Branch
BIC	Bit Clear
BKPT	Software breakpoint
BL	Branch with Link
BLX	Branch with Link, change instruction set
BX	Branch, change instruction set
CMN, CMP	Compare negative, compare
CPSID	Disable interrupts

(continued)

S. Smith, *RP2040 Assembly Language Programming*,
https://doi.org/10.1007/978-1-4842-7753-9

Instruction	Description
CPSIE	Enable interrupts
DMB, DSB	Data Memory Barrier, Data Synchronization Barrier
EOR	Exclusive OR
ISB	Instruction Synchronization Barrier
LDM	Load multiple registers
LDR	Load register with word
LDRB	Load register with byte
LDRH	Load register with halfword
LDRSB	Load register with signed byte
LDRSH	Load register with signed halfword
LSL, LSR	Logical shift left, logical shift right
MOV	Move
MRS	Move from PSR to register
MSR	Move from register to PSR
MUL	Multiply
NEG	Two's complement
NOP	No operation
ORR	Logical OR
PUSH, POP	PUSH registers to stack, POP registers from stack
REV	Reverse bytes in word
REV16, REVSH	Reverse bytes in halfword
ROR	Rotate right register
SBC	Subtract with carry

(continued)

Instruction	Description
SEV	Set event
STM	Store multiple registers
STR	Store register with word
STRB	Store register with byte
STRH	Store register with halfword
SUB	Subtract
SVC	Supervisor call
SXTB, SXTH	Signed extend
TST	Test
UXTB, UXTH	Unsigned extend
WFE, WFI	Wait for event, wait for interrupt
YIELD	Yield

Answers to Exercises

Here, we have answers to selected exercises. For program code, check the online source code at the Apress GitHub site.

Chapter 2

2-1. 0100 1101 0010, 0x4d2

Chapter 4

4-1. 177 (0xb1), 233 (0xe9)
4-2. -14, -125
4-3. 0x78563412
4-4. 0x118
4-5. 0x218

Chapter 6

6-2. The **LDR** instruction either provides an offset to the **PC** directly from the address or creates the address in the code section using indirection from the **PC** to load this value.

S. Smith, *RP2040 Assembly Language Programming*,
https://doi.org/10.1007/978-1-4842-7753-9

Chapter 9

9-1. 0x40044000, i2c.h

9-2. The more pins, the larger the size of the board. This is a trade-off to keep the board small but still provide a great deal of flexibility.

Chapter 10

10-1. 65104

10-2. 62,500,000 Hz or 62.5MHz

Index

S. Smith, *RP2040 Assembly Language Programming*,
https://doi.org/10.1007/978-1-4842-7753-9

I

J, K

L

M

N

O

P, Q

R

S