MAKER INNOVATIONS SERIES

RP2040 Assembly Language Programming

Including the RP2350 and Raspberry Pi Pico 2

Second Edition

Stephen Smith

Apress®

Stephen Smith

RP2040 Assembly Language Programming
Including the RP2350 and Raspberry Pi Pico 2

Second Edition

Apress®

Stephen Smith
Gibsons, BC, Canada

ISSN 2948-2542 e-ISSN 2948-2550
Maker Innovations Series
ISBN 979-8-8688-2201-8 e-ISBN 979-8-8688-2202-5
https://doi.org/10.1007/979-8-8688-2202-5

This Apress imprint is published by the registered company APress Media, LLC, part of Springer Nature.

The registered company address is: 1 New York Plaza, New York, NY 10004, U.S.A.

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

Introduction

There is an explosion of DIY electronics projects, largely fueled by 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 design robots, home monitoring and security systems, game devices, musical instruments, audio systems, and lots 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. The RP2040 chips can even be purchased for approximately $1 each to build your own board.

At the basic level, how are these microcontrollers programmed? What provides the magical foundation for all the great projects that programmers build with them? Raspberry provides a Software Developer’s Kit (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 multi-core processor, programmable I/O processors, and integrated hardware controllers. Anything learned about these devices is directly relevant to any gadget with an ARM processor that by volume is the number one processor on the market today.

In this book, how to program ARM Cortex-M0+ processors at the lowest level, operating as close to the hardware as possible, is covered. How to do the following will be learned:

- Format instructions and combine them into programs, as well as the formats of operative binary data.
- Program the built-in programmable I/O, division, and interpolation coprocessors.
- Control the integrated hardware devices by reading and writing to the hardware control registers directly.
- 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 day-to-day life, knowing how the processor works at the Assembly Language level and the low-level binary data structures will make for better programming in all other areas. Knowing how the processor works will translate to writing more efficient C code and can even help with Python programming.

Enjoy this introduction to Assembly Language. Learning it for one processor family helps with learning and using any other processor architectures encountered throughout a programmer's career.

Introduction to the Second Edition

Since the release of the first edition, the Raspberry Pi Organization has released an updated version of their custom chip, namely, the RP2350. Then, based on the RP2350 is the Raspberry Pi Pico 2. As a result, this book often refers to the Pico-series to cover both the Pico 1 and Pico 2. The RP2350 is based on the newer ARM Cortex M33, which runs faster, has more memory, and includes a single-precision floating-point unit.

Over the same time period, Raspberry has updated their C/C++ SDK, with the most notable addition being to the Visual Studio (VS) Code extension—a popular and productive way to develop software for the Raspberry Pi Pico-series. Some of the main features in this addition include

1. Instructions to using Visual Studio Code
2. How to program the RP2350 floating-point unit
3. How to use some advanced M33 instructions like the division instructions
4. How to enable the pads for the RP2350 due to the new electrical isolation feature
5. How to perform debugging via the Raspberry Pi Debug Probe

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-Second-Edition`

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

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 Stewart Watkiss, my technical reviewer, who helped make this a far better book.

Table of Contents

About the Author

Stephen Smith
is 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; 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 smist08.wordpress.com and has written two science fiction novels in a series, *Influence* and *Unification*, available on 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 computer to school children 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`).

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
S. Smith, *RP2040 Assembly Language Programming*, Maker Innovations Series
https://doi.org/10.1007/979-8-8688-2202-5_1

1. How to Set Up the Development Environment

Stephen Smith[1]
(1) Gibsons, BC, Canada

This chapter is concerned with physically setting up the Raspberry Pi Pico 2 on a breadboard and wiring it up to a host computer to effortlessly program and debug programs, as well as hooking up other components as they're encountered. The *Getting started with Raspberry Pi Pico-series* guide (from the Raspberry Pi Organization's website) is an excellent reference on how to do these fundamental tasks. That content is not duplicated here; instead, the important parts that are required for Assembly Language programming are pointed out to debug and play with the sample programs in this book.

To run most of the programs in this book, the following equipment is needed:

- A Raspberry Pi Pico (1 or 2)
- A Raspberry Pi Debug Probe
- An electronics breadboard
- Pins to attach the Pico to the breadboard
- Miscellaneous connecting wires
- A selection of LEDs
- A soldering iron and solder or an “H” series Pico
- A Raspberry Pi 4 or 5 running the Raspberry Pi OS

About the Pico Families

Microcontrollers like the Raspberry Pi Pico 2 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 microcontrollers usually don’t have a keyboard, mouse, or monitor, their programs are developed on a regular computer, known as a host computer, and then uploaded to the microcontroller to test and finally deploy them.

The Raspberry Pi Organization has two families of microcontrollers:

- **The Pico 1 family**: Built around Raspberry’s RP2040 ARM CPU
- **The Pico 2 family**: Built around Raspberry’s RP2350 ARM CPU

Each of these families consists of various models, where a “W” after the name indicates wireless support including Wi-Fi and Bluetooth and an “H” after the name indicates pre-soldered headers. The four digits after the RP indicate the number and type of CPU cores along with the amount of memory.

When reading this book, there could well be additional members in this family of processors; however, most of the content will apply to these as well.

Not only are the RP2040 and RP2350 chips the heart of the Raspberry Pi Pico families, but Raspberry also sells these chips to other manufacturers, including Adafruit, Arduino, Seeed Studio, SparkFun, and Pimoroni. These other companies produce boards like the Raspberry Pi Picos but with different feature sets, for instance, in different form factors or with different connectors to easily integrate into other modular systems.

In this book, when the RP2040 or RP2350 is referred to, it applies to all the brands of RP2040- or RP2350-based boards. However, in some cases a specific board is talked about to discuss Wi-Fi or a specific wiring connection for one board.

Note The RP2350 contains two RISC-V CPU cores in addition to the two ARM cores. This book covers how to program the ARM CPUs in ARM Assembly Language. The RISC-V cores are a different Assembly Language and require a separate book such as *RISC-V Assembly Language Programming* by Stephen Smith also from Apress.

Programming the RP2040 or RP2350 in Assembly Language is the main emphasis of this book, but this is best done by studying real working programs. To do this, the microcontroller needs to be connected to various pieces of hardware. This way programs that perform useful tasks can be seen, and all the flexible and powerful features of the RP2040/RP2350 can be learned including how to connect to external sensors, controllers, and communication channels. To begin, a Raspberry Pi Pico 2 is set up on an electronics breadboard, so it can easily be wired to various devices.

About the Raspberry Pi Pico-series

The heart of the Raspberry Pi Pico-series is a chip developed by Raspberry and ARM. There are now two flavors of this chip, the older RP2040 and the newer RP2350. Each chip is a System on a Chip (SoC) that contains dual-core ARM Cortex CPUs, SRAM, a USB port, and support for several hardware devices. Compared with a full computer like the regular Raspberry Pi, the Raspberry Pico-series lacks a video output port, an operating system, and connectors for a keyboard and mouse. But it is possible to connect displays and input devices to the Raspberry Pi Pico through its GPIO pins. The specialty connections and input devices aren't used for general-purpose computing; rather, they solve specific problems, such as powering a vending machine or monitoring a greenhouse.

Unlike the CPUs found in desktop and laptop computers, the RP2040/RP2350 doesn't support advanced modules like a vector processing unit, a virtual memory controller, or a 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. These are covered in Chapter 10. If a Pico-series board is already wired up and how to download and debug C programs is understood, then skip ahead to Chapter 2.

The RP2040/RP2350 may look underpowered when comparing it with a modern Intel, AMD, or ARM processor, but for the price it is quite a powerful computer. Table 1-1 compares the RP2040 and RP2350 with some older and newer computers as well as competitors' microcontrollers.

Table 1-1 Comparison of the processing power of the RP2040 and RP2350

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 R4	ARM M4	48	32	32	1
Arduino Due	ARM M3	84	96	32	1
RP2040	**ARM M0+**	**133**	**264**	**32**	**2**
RP2350	**ARM M33**	**150**	**520**	**32**	**2**
Pi Zero	ARM A53	1024	524,288	32	1
Pi 5	ARM A76	2400	16,777,216	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/RP2350- based microcontrollers, this could be on a MacOS, Windows, or Linux-based computer. The Raspberry Pi Pico-series documentation has instructions on how to connect them to all these platforms. The easiest solution is to use a Raspberry Pi 5 as the host versus 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 5 and Raspberry Pi Pico-series together.

About the Raspberry Pi Debug Probe

USB ports are a wonderful invention because they allow all sorts of devices to be easily connected. Raspberry Pi Pico-series boards have a USB port that connects to the host computer. This permits programs to be downloaded to the Pico-series board to run and lets messages return to the host computer. This is great, but there is a problem when a program needs to be debugged. For a USB connection to work properly, the CPU must continually communicate with it, or the device becomes disconnected. When debugging a program, the debugger needs to stop the program executing so that the registers and memory can be examined. On full computers with multi-tasking operating systems, this isn't a problem as other programs maintain the USB ports while the program being developed is debugged. But beware that on the Raspberry Pi Pico-series board, there is only one program running, so when the debugger stops this program, it stops everything on the board.

The solution to this problem is to not use the USB port for debugging; instead, there are separate debug pins for the debugger to control the board and serial communications via a UART for messaging. The serial communications ports don't require continual attention, so don't react when the CPU is stopped. In the first edition of this book, this required wiring pins from a Raspberry Pi 4's GPIO pins to the Raspberry Pi Pico's debug pins and UART pins. This was cumbersome and presented a problem for programmers wanting to use a regular Windows, MacOS, or Linux computer as their host computer.

To solve this problem, Raspberry invented the Raspberry Pi Debug Probe. This is a device containing an RP2040 chip running a custom program whose job is to mediate between the Raspberry Pi Pico-series debug and UART pins and the host computer's USB port. The processor on the Debug Probe keeps the USB connection alive and translates the data between the host's USB connection and the Pico's debug and serial ports. With this new feature, it's easy to develop and debug programs from any Windows, MacOS, or Linux-based computer for the Raspberry Pi Pico-series. This makes wiring up the Pico-series board to a Raspberry Pi 5 easier as well.

This book will assume the use of a Raspberry Pi Debug Probe; however, feel free to follow Raspberry's instructions for other possible solutions.

How to Solder and Wire

It is possible to get by without doing any soldering if purchasing the "H" version of the Raspberry Pi Pico 1 or 2, which has header pins pre-soldered to the board making it ready to press into an electronics breadboard. Similarly, with a Raspberry Pi Debug Probe, soldering to the debug pins can sometimes be avoided. One way or another, the Raspberry Pi Pico needs to be connected to external devices. Without this, programs can be downloaded to the Pico-series, the onboard LED can be flashed, and data can be sent back out the USB port to the host computer. However, even to debug a program, some external connections are required.

Often the "H" series Picos with the pre-soldered headers are sold out or are sometimes a bit expensive just to save some soldering. The easiest way to experiment with a Pico is to have it connected to an electronics breadboard, which requires headers attached to the board either included or hand soldered.

Typically, a new Pico-series board would be soldered into a final project directly. At $4 each (at the time of this writing), there isn't a significant overhead in having a development board and adding new boards to the package when finished. To perform debugging requires soldering pins to the three debugging connections on the end of the board, if they didn't come pre-installed. The minimum wiring needed are the following four connections between the Pico and the Raspberry Pi 5:

1. A micro-USB cable connecting the Pico-series to the host computer

2. A micro-USB cable connecting the Raspberry Pi Debug Probe to the host computer
3. The three Pico-series debug pins connected to the Raspberry Pi Debug Probe
4. Three pins connecting a serial port on the Pico-series to the Raspberry Pi Debug Probe

Please refer to *Getting started with Raspberry Pi Pico-series* for the full details.

Don't fear soldering; it is quite simple and fun. The main trick is to heat up the area where the solder should 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. Figure 1-1 shows the wiring of a Raspberry Pi Pico 2 connected to a Raspberry Pi Debug Probe. The two USB cables then connect to the Raspberry Pi 5 host computer.

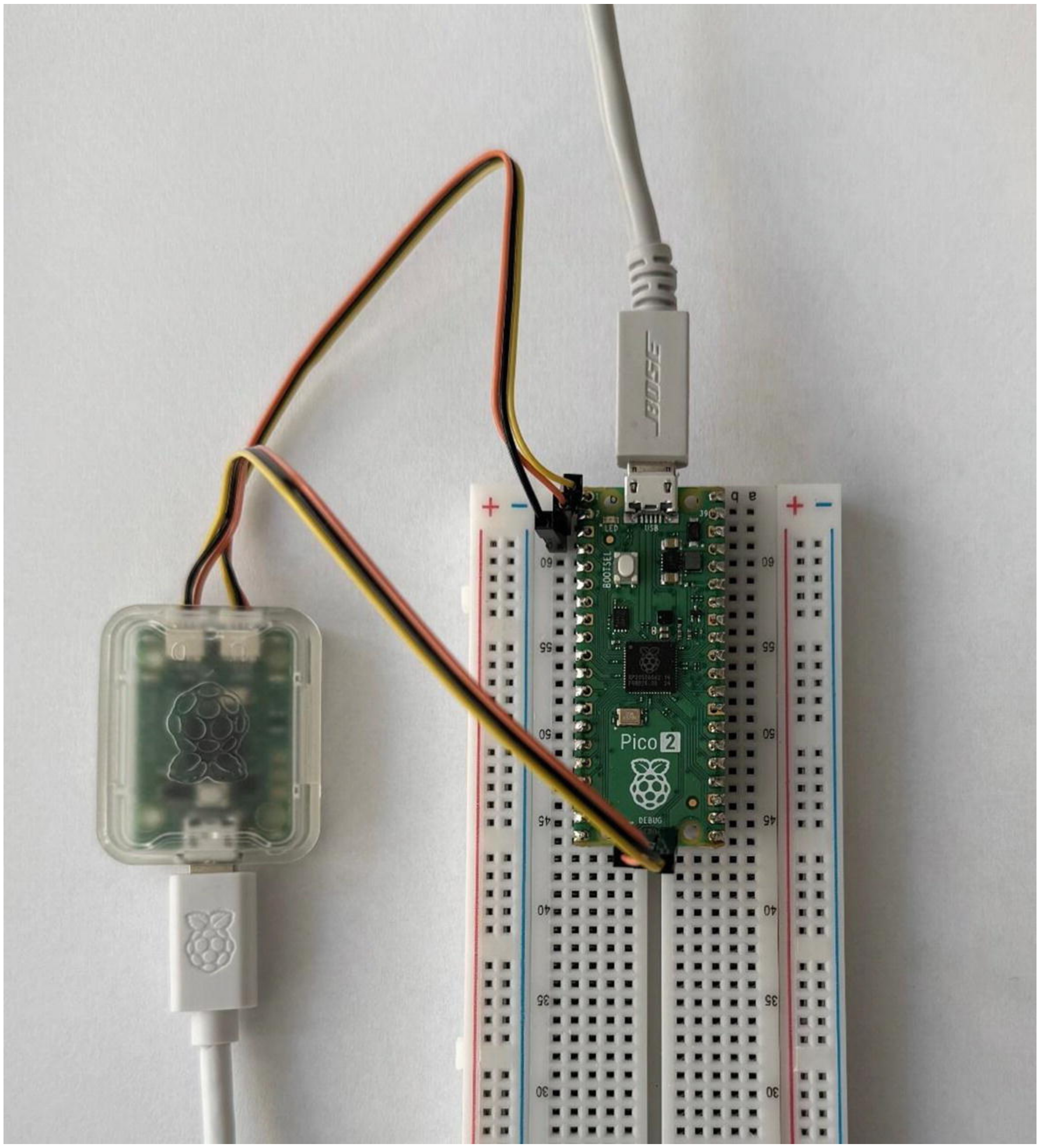


Figure 1-1 A Raspberry Pi Pico 2 installed in a breadboard and connected to a Raspberry Pi Debug Probe. The USB cables connect to the Raspberry Pi 5 host computer

> **Note** If using an RP2040/RP2350 board from another vendor, then it is likely that the pins are in different locations, and the wiring needs to be adapted for the location of the pins.

How to Install Software

If using a Raspberry Pi with the Raspberry Pi OS as the host computer, then this is straightforward. This simplifies installation, since it runs 32-bit ARM code and shares development tools with the Raspberry Pi Pico-series and other RP2040/RP2350-based boards.

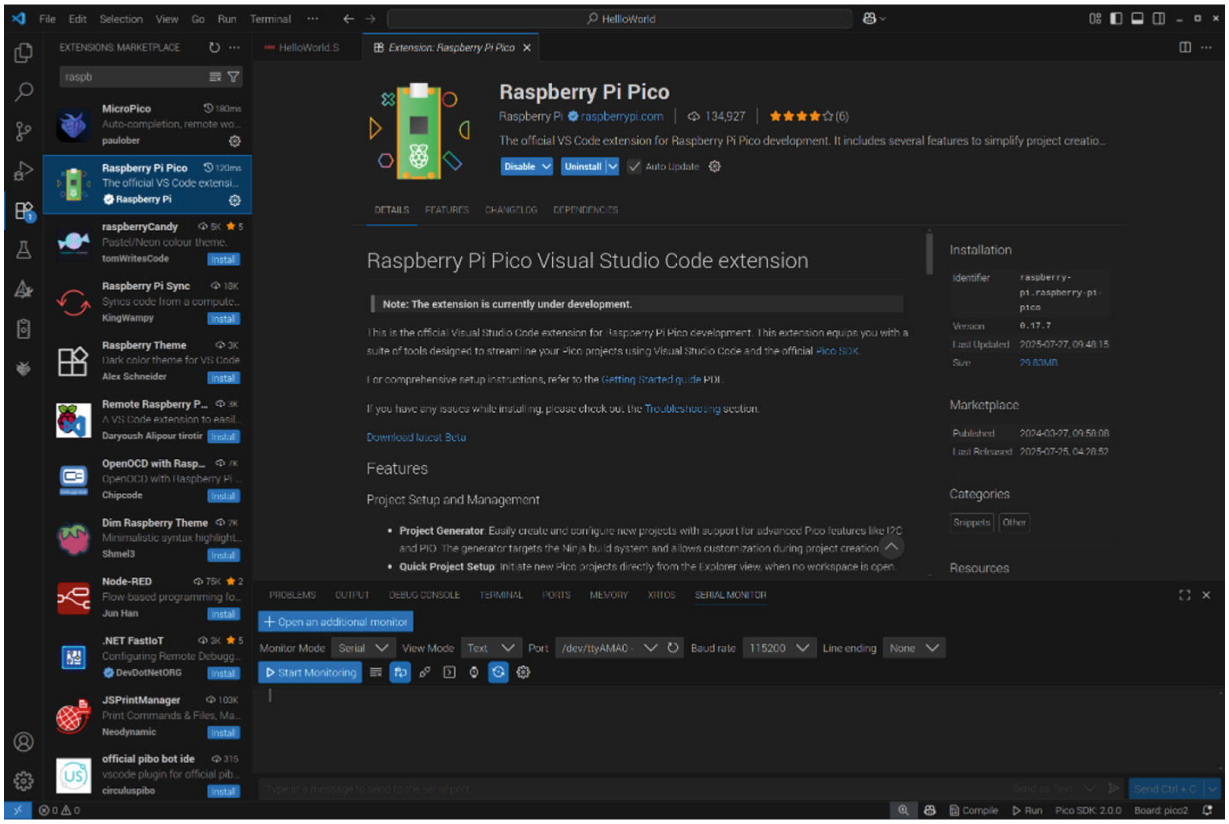
The *Getting started* guide includes instructions for working with Visual Studio Code. The easiest way to get up and running is to install Visual Studio Code and to add the Raspberry Pi Pico-series extension, which installs everything needed. This book also covers installing the Pico-series SDK separately and then working with text files that can be edited in any editor, using **cmake** and **make** for building, **gdb** (GNU debugger) and **openocd** for debugging, and **minicom** for communications.

Using Visual Studio Code

To install Visual Studio Code and a few other required dependencies, use the following commands:

```
sudo apt update
sudo apt install code
```

Now run Visual Studio Code; find the extensions marketplace, as shown in Figure 1-2; and install the Raspberry Pi Pico code extension.


Raspberry Pi Pico

Raspberry Pi raspberrypi.com | 134,927 | ★★★★☆ (6)

The official VS Code extension for Raspberry Pi Pico development. It includes several features to simplify project creatio...

Raspberry Pi Pico Visual Studio Code extension

Note: The extension is currently under development.

This is the official Visual Studio Code extension for Raspberry Pi Pico development. This extension equips you with a suite of tools designed to streamline your Pico projects using Visual Studio Code and the official Pico SDK.

For comprehensive setup instructions, refer to the Getting Started guide PDF.

If you have any issues while installing, please check out the Troubleshooting section.

Download latest Beta

Features

Project Setup and Management

- **Project Generator**: Easily create and configure new projects with support for advanced Pico features like I2C and PIO. The generator targets the Ninja build system and allows customization during project creation.
- **Quick Project Setup**: Initiate new Pico projects directly from the Explorer view, when no workspace is open.

Installation

Identifier	raspberry-pi.raspberry-pi-pico
Version	0.17.7
Last Updated	2025-07-27, 09:48:15
Size	29.83MB

Marketplace

Published	2024-03-27, 09:58:08
Last Released	2025-07-25, 04:28:52

Categories

Snippets Other

Resources

Monitor Mode: Serial View Mode: Text Port: /dev/ttyAMA0 Baud rate: 115200 Line ending: None

Figure 1-2 The Raspberry Pi Pico extension in the Visual Studio Code marketplace

Installing the Full SDK

The Visual Studio Code extension places a minimal copy of the Pico-series SDK under each project. This is required since any parts of the SDK used in the project need to be compiled into the final executable. However, the full SDK can be installed into a central location and then shared by various separate projects. If the host computer is a Raspberry Pi, this is simple to set up as there is a shell script that can be run to do the whole job. “Appendix C: Manual toolchain setup” in the *Getting started with Raspberry Pi Pico-series* guide explains this process and provides the necessary steps.

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-series* book has a walk-through on how to do this. Here, rather than duplicate, a list of the key things needed throughout this book is provided. Using either Visual Studio Code or the standalone SDK is fine, though Visual Studio Code automates some of these processes. Here's a list of the prerequisite skills:

- How to load and run a program. Visual Studio Code will either do this automatically or give an error that the Pico needs to be powered on and off holding down the **bootsel** button. Using the standalone SDK the Pico needs to be powered off and on while holding down the **bootsel** button, and then the program must be manually copied to the shared drive.
- How to compile a program to either send its output to the USB or serial port. Visual Studio Code has a dialog to specify this; for the standalone SDK, the **CMakeLists.txt** file needs to be edited.
- How to display output from the Pico either using Visual Studio Code's monitor pane or using minicom to display the output that the Pico is sending.
- How to compile a program for debug.
- How to debug a program, either using Visual Studio Code or using **openocd** and **gdb**.

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

Create Some Helper Script Files

When following along with the *Getting started with Raspberry Pi Pico-series* guide, there are many long command lines to type in (or to copy/paste). It saves significant time to create a collection of small shell scripts to automate the common tasks. These can be placed in $HOME/bin. Then add

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

to the end of the $HOME/.bashrc file. These all need to be made executable with

```
chmod +x filename
```

First, a script for minicom—one to listen on either the USB or UART for text messages:

File m-usb:

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

File m-uart:

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

Note If using the Raspberry Pi Debug Probe, then **m-usb** will always be used as the Debug Probe's job is to turn UART traffic into USB traffic.

To build debug, a helpful script is **cmaked** containing

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

To ensure **openocd** is fully installed, run

```
sudo apt install openocd
```

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

```
sudo openocd -f interface/cmsis-dap.cfg -f
```

is helpful. This version is for using the Raspberry Pi Debug Probe and a Raspberry Pi Pico 2. If debugging via a different method or using a different board, then different **cfg** files are required. Note that all the **cfg** files to choose from are found in the two folders:

```
$HOME/.pico-sdk/openocd/0.12.0+dev/scripts/target
$HOME/.pico-
sdk/openocd/0.12.0+dev/scripts/interface
```

When **gdb** starts, it needs to connect to **openocd**. This can be automated by creating a **.gdbinit** file in the $HOME folder. This file then contains

```
target remote localhost:3333
```

> **Note** This **.gdbinit** will be used anytime **gdb** is started, so if debugging a local file without using **openocd** is needed, then this file needs to be renamed while this is done.

Summary

This chapter is the starting point for programming the Raspberry Pico-series board. No Assembly Language programming has been done yet, but now the development environment is set up to write, debug, test, and deploy programs written in either C or Assembly Language. The host computer, say a Raspberry Pi 5, is connected to the Raspberry Pi Pico-series board through a USB cable and to the Raspberry Pi Debug Probe through a second USB cable. The Raspberry Pi Pico-series board is connected to the Raspberry Pi Debug Probe through its serial port pins and the debugging port pins. The Pico-series board is installed in an electronics breadboard ready to have other components connected to it. In Chapter 2, all these tools will be used to start the journey to writing a program with RP2040/RP2350 Assembly Language.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
S. Smith, *RP2040 Assembly Language Programming*, Maker Innovations Series
https://doi.org/10.1007/979-8-8688-2202-5_2

2. The First Assembly Language Program

Stephen Smith[1]
(1) Gibsons, BC, Canada

Most of the functionality of a Raspberry Pi Pico-series is contained in the Raspberry-designed custom RPxxxx chip such as the RP2040 or RP2350. These contain dual-core ARM Cortex-M-series CPUs such as the Cortex-M0+ or Cortex-M33. 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. Unhappy with the microprocessors that were around at the time, since they were much more complicated than the 6502, and not wanting 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 it's at a good volume, manufacturing is costly. 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 devices that ran for a long time on a single battery charge.

Unlike Intel, ARM doesn't manufacture chips; it 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 are the Cortex-M-series. This book concentrates on the ARM Cortex M0+ used in Raspberry Pi's RP2040 SoC and the ARM Cortex-M33 used in the RP2350 SoC.

To make these chips inexpensive, the transistor count is reduced as much as possible. The M-series CPUs are all 32-bit and 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.

This book examines how the Cortex-M-series 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. The ones included in the RP2040 and RP2350, such as the fast integer multiplier and divider (multiplication and division are an extra), are considered. The RP2350 even includes a floating-point unit.

Ten Reasons to Use Assembly Language

A Raspberry Pi Pico-series chip can be programmed with MicroPython or C/C++. These are productive languages that hide the details of all the bits and bytes, keeping the focus on the application. When programming in Assembly Language, the program is tightly coupled to the current CPU, and moving the 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 Language is harder, as problems with memory addressing and CPU registers must be solved, which are all handled transparently by high-level languages. So why would a programmer ever want to learn Assembly Language programming? Here are ten reasons people learn and use Assembly Language:

1. **To write more efficient code**: Knowing how the computer works internally leads to writing more streamlined code, even if never writing Assembly Language code. For example, make data structures easier to access and write code in a style that allows the compiler to generate more effective code. Also, make better use of computer resources, like coprocessors, and use the given computer to its fullest potential.
2. **To utilize specialty coprocessors**: The PIO coprocessors on the RP2040/P2350 are only programmable in Assembly Language. There is a library of common applications in the Software Developer's Kit (SDK), but if something beyond these is needed, Assembly Language is the only option.
3. **To more effectively debug programs**: When debugging any program on the Pico-series using **gdb**, a lot of the views are at the Assembly Language level. The Assembly Language code generated by the compiler can be seen, along with the CPU registers and raw memory. Understanding this extra level of detail can help solve more difficult program bugs. Further, much of the SDK is written in Assembly Language, and understanding it helps when stepping through the code.
4. **To make RP2040/RP2350 programs 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. **To improve hardware interfaces**: When interfacing the Pico-series to a hardware device through the GPIO ports, 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 more efficient to program in Assembly Language.
6. **To improve Machine Learning**: The Pico-series is fast enough to perform Machine Learning. This relies on fast matrix mathematics. If this can be made faster with Assembly Language and/or using the coprocessors, then an AI-based robot or sensor network can be made that much better.
7. **To optimize specific functions**: Most large programs have components written in different languages. If the program is 99% C++ and Python, the other 1% could be Assembly Language, perhaps giving the program a performance boost or some other competitive advantage.
8. **To add hardware support to the SDK**: If working for a hardware company

that makes an RP2040/RP2350-based board competitor to the Raspberry Pi Pico-series, these boards have some Assembly Language code in the SDK that must be customized for their specialized functionality or configuration.

9. **To look for security vulnerabilities**: When searching for security vulnerabilities, the Assembly Language code needs to be examined; otherwise, there isn't a way to know what is really going on and hence where holes might exist. This is especially important when connecting to IoT networks.

10. **To conserve precious resources**: When programming microcontrollers, there are limited memory and resources. Often every bit needs to be used efficiently to get an application to do what is needed. Often Assembly Language is the only option to cram in every bit of functionality possible.

Computers and Numbers

Numbers for humans are typically represented using base 10. The common theory is that this is done because humans have ten fingers to count with. This means a number like 387 is a representation for

$$\begin{aligned} 387 &= 3 * 10^2 + 8 * 10^1 + 7 * 10^0 \\ &= 3 * 100 + 8 * 10 + 7 \\ &= 300 + 80 + 7 \end{aligned}$$

There is nothing special about using 10 as the 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 humans have 20 digits—10 fingers and 10 toes.

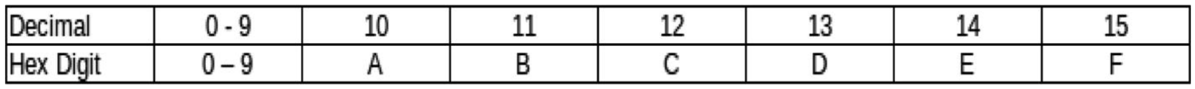
Computers don't have fingers or toes; rather, everything is a switch that is either on or off. As a result, it is natural for computers to use base 2 arithmetic. Thus, to a computer a number like 1011 is represented by

$$\begin{aligned} 1011 &= 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 \\ &= 1 * 8 + 0 * 4 + 1 * 2 + 1 \\ &= 8 + 0 + 2 + 1 \\ &= 11 \text{ (decimal)} \end{aligned}$$

This is great for computers, but four digits are used 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 is the byte, that is, 8 binary bits or digits. In the example above with 4 bits, there are 16 possible combinations of 0s and 1s. This means 4 bits can represent the numbers 0–15. This means each number can be represented by one base 16 digit. Base 16 digits are represented by the numbers 0–9 and then the letters A–F for 10–15. Base 16 numbers are referred to as hexadecimal (Figure 2-1).



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

Figure 2-1 Representing hexadecimal digits

A byte (8 bits) can be represented as two base 16 digits. 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 (decimal)
   = 1110 0110 (binary)
```

An ARM Cortex-M processor handles 32-bit numbers, a 32-bit quantity is called a word, and it is represented by 4 bytes. So a string like B6 A4 44 04 is seen 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. It's just a matter of understanding what is presented on screen. Also, if an exact binary number needs to be specified, usually that is done 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 several computer-related logical operations. Figure 2-2 is a screenshot of this calculator displaying the hex number E6.

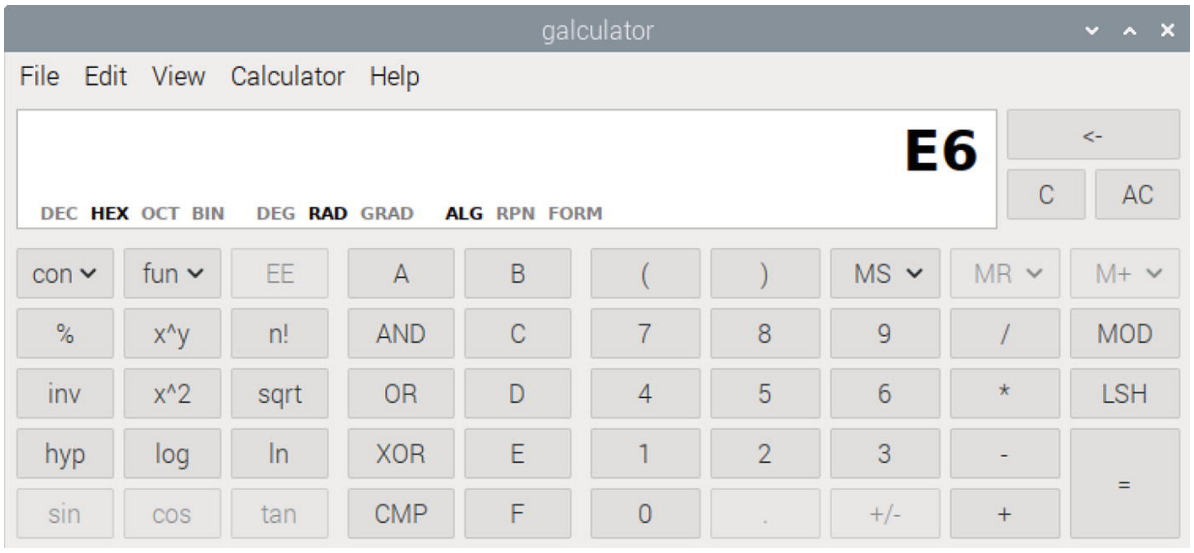

galculator
File Edit View Calculator Help
E6
DEC HEX OCT BIN DEG RAD GRAD ALG RPN FORM
<-
C AC
con fun EE A B ( ) MS MR M+
% x^y n! AND C 7 8 9 / MOD
inv x^2 sqrt OR D 4 5 6 * LSH
hyp log ln XOR E 1 2 3 - =
sin cos tan CMP F 0 . +/- +

Figure 2-2 The Raspberry Pi OS's galculator

There is a bit more complexity in how signed integers are represented and how arithmetic works. This is explained later when arithmetic is covered.

ARM Assembly Instructions

In this section, the basic architectural elements of the ARM Cortex-M0+ processor are introduced, and the form of its machine code instructions is looked at. The ARM processor is a Reduced Instruction Set Computer (RISC), which theoretically will make learning Assembly Language 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 easy tasks. The goal is to develop design patterns to help building more sophisticated structured programming elements.

When programming an ARM A-series CPU, in 32-bit mode, as with the Raspberry Pi 5, then there is a subset of the instruction set called the "thumb" instructions. Newer A-series CPUs typically have 32-bit instructions, but if memory needs to be conserved, then there is a "thumb" mode. When switching 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. This book won't continue to refer to them as thumb instructions, since these are the full instruction set of the Cortex-M-series CPUs used in the RP2040 and RP2350. Running a simpler instruction set is a key design decision to keep the transistor count down; therefore, the cost and power consumption of M-series processors are 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 CPU registers, and then the data processing or arithmetic operation is performed in these 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.

To add two numbers:

1. Load one into one register and the other into another register.
2. Perform the add operation putting the result into a third register.
3. Copy the answer from the results register back to memory.

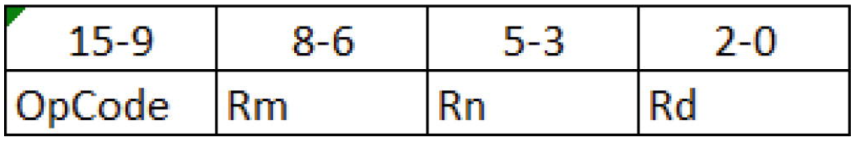
This is typical of a RISC processor where it takes several instructions to perform simple operations.

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

- **R0–R7**: These eight are general-purpose that can be used for anything.
- **R8–R11**: These registers can be used to store values, but there are few instructions that can access these directly.
- **R12**: The intra-procedure call scratch register (**IP**)**.**
- **R13**: The stack pointer (**SP**).
- **R14**: The link register used in the context of calling functions, which will be explained in more detail when subroutines are covered.
- **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 branch instructions (if statements) are covered.

ARM Instruction Format

Most ARM Cortex-M-series binary instructions are 16 bits long. There are a small number of 32-bit-long instructions that will be talked about when encountered. 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 several instruction formats, and these will be explained when they are encountered. To give an idea for some data processing instructions, consider the format for an ADD instruction. Figure 2-3 is the format of the instruction and what the bits specify.



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

Figure 2-3 The binary format of the ADD instruction

Examining each of these fields:

- **Opcode**: Which instruction is being performed, 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 following Assembly Language instruction:

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

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

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 the use of 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 a register won't be specified.

If familiar with A-series Assembly Language, this instruction is actually **ADDS**, since it "sets" the **CPSR** when it executes. M-series Assembly Language doesn't have the option to control whether the **CPSR** is set, so it tends to be left off; however, the Assembler will take either.

In A-series Assembly Language, this instruction can be seen as **ADD.N,** meaning narrow, indicating the 16-bit encoding instead of **ADD.W,** which 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 there is a linear block of instructions, they all execute on average taking one clock cycle.

RP2040/RP2350 Memory

The RP2040 has 264 kilobytes (kb) of memory, and the RP2350 has 520kb of memory. Programs are loaded from the Pico-series' 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 an ARM M-series CPU 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.

To load a register from a known 32-bit memory address, for example, a variable to perform arithmetic on, is a common operation. How is this done? The instruction is only 16 bits in size, and nearly all the bits are already used to specify the opcode and register to use.

This is a problem that will be returned 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.

Memory can be loaded by using a register to specify the address to load. This is called indirect memory access. But all this does is move the problem, since there still isn't 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 allows efficient access memory within 256 words of the PC. Yuck, how would a programmer write such code? This is where the GNU Assembler comes in. It allows the location to be specified symbolically and will figure out the offset automatically.

In Chapter 6, the details of accessing memory will be studied in detail. In all RISC processors this is a challenge since the size of memory addresses is typically larger than the size of the Assembly Language instructions.

About the GCC Assembler

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

```
label:     opcode    operands
```

The **label**: is optional and only required if the instruction is 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

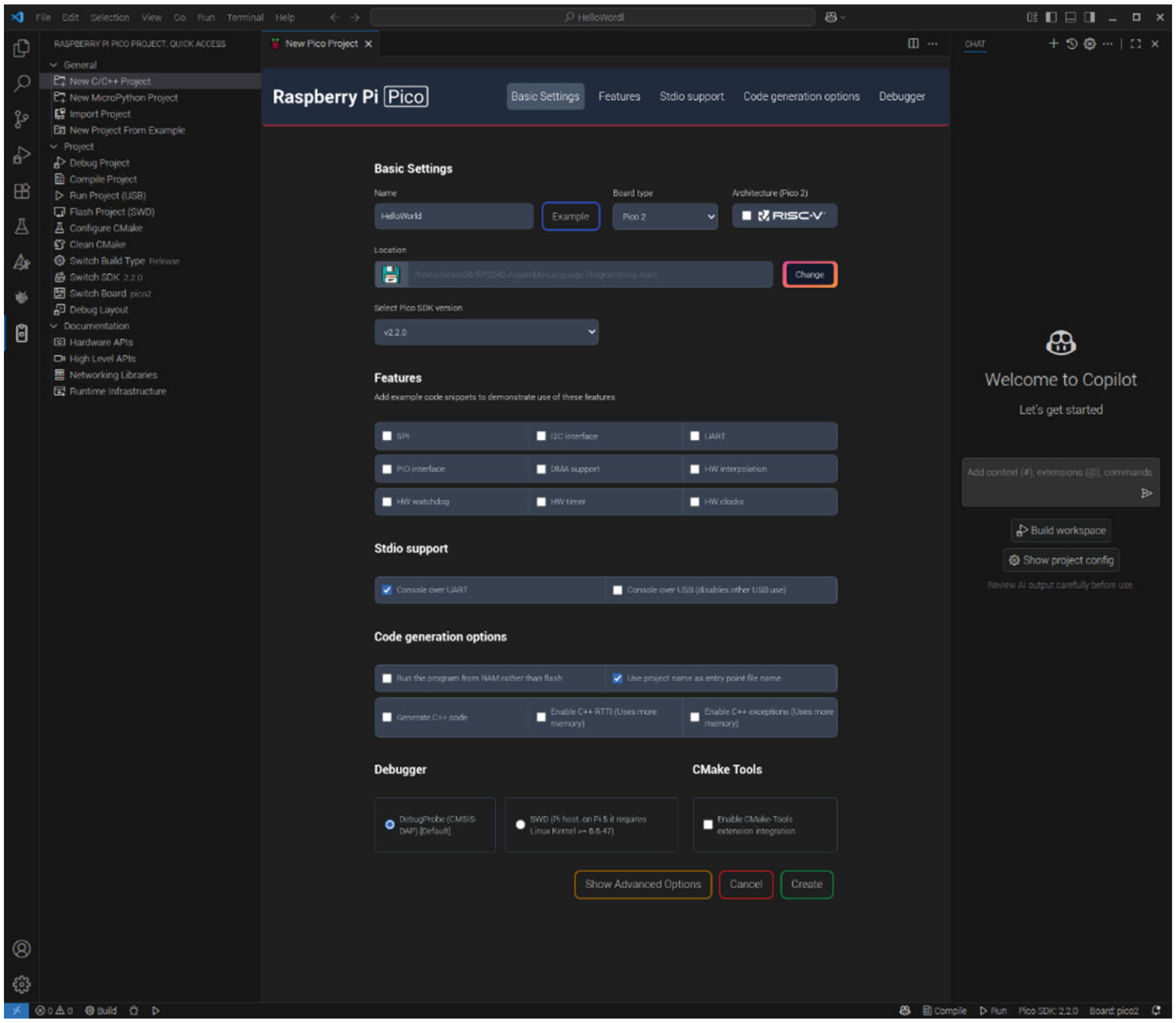
There are quite a few different formats for the operands, and these will be covered as the instructions that use them are encountered.

Hello World

In almost every programming book, the first program is a simple program to output the string "Hello World". This will now be done with Assembly Language to demonstrate some of the concepts discussed. This sample will be built both with Visual Studio Code and the raw Pico-series C/C++ SDK framework, to demonstrate the two common ways of building projects. Up first is Visual Studio Code.

With Visual Studio Code

Start up Visual Studio Code and, from the Raspberry Pi Pico extension, choose to create a new C/C++ project. Figure 2-4 shows the New Pico Project screen with the necessary fields filled in when using the Raspberry Pi Debug Probe:



Basic Settings

Name: HelloWorld | Example | Board type: Pico 2 | Architecture (Pico 2): ☐ RISC-V

Location: /home/smist08/RP2040-Assembly-Language-Programming-main | Change

Select Pico SDK version: v2.2.0

Features

Add example code snippets to demonstrate use of these features

☐ SPI ☐ I2C interface ☐ UART
☐ PIO interface ☐ DMA support ☐ HW interpolation
☐ HW watchdog ☐ HW timer ☐ HW clocks

Stdio support

☑ Console over UART ☐ Console over USB (disables other USB use)

Code generation options

☐ Run the program from RAM rather than flash ☑ Use project name as entry point file name
☐ Generate C++ code ☐ Enable C++ RTTI (Uses more memory) ☐ Enable C++ exceptions (Uses more memory)

Debugger

● DebugProbe (CMSIS-DAP) [Default] ○ SWD (Pi host, on Pi 5 it requires Linux Kernel >= 6.6.47)

CMake Tools

☐ Enable CMake-Tools extension integration

Show Advanced Options | Cancel | Create

Figure 2-4 The Visual Studio Code Raspberry Pi Pico extension New Pico Project dialog

- **Name**: HelloWorld
- **Stdio support**: Console over UART
- **Debugger**: DebugProbe (CMSIS-DAP)

Now take the code from Listing 2-1 and copy it to a file called **HelloWorld.S** placed in the folder specified for the HelloWorld project.

Note It is important to use .S and not .s in the filename, because .S will support some C-type include files, whereas .s is for pure Assembly Language only. As more of the SDK is used, more C-type files will need to be included.

```
@
@ 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"
```

Listing 2-1 The Hello World program

Now delete the **HelloWorld.c** file that was created in that same folder. This work will automatically be represented in the file list for the project. However, the CMakeLists.txt file needs to be edited to change

```
add_executable(HelloWorld HelloWorld.c )
```

to

```
add_executable(HelloWorld HelloWorld.S )
```

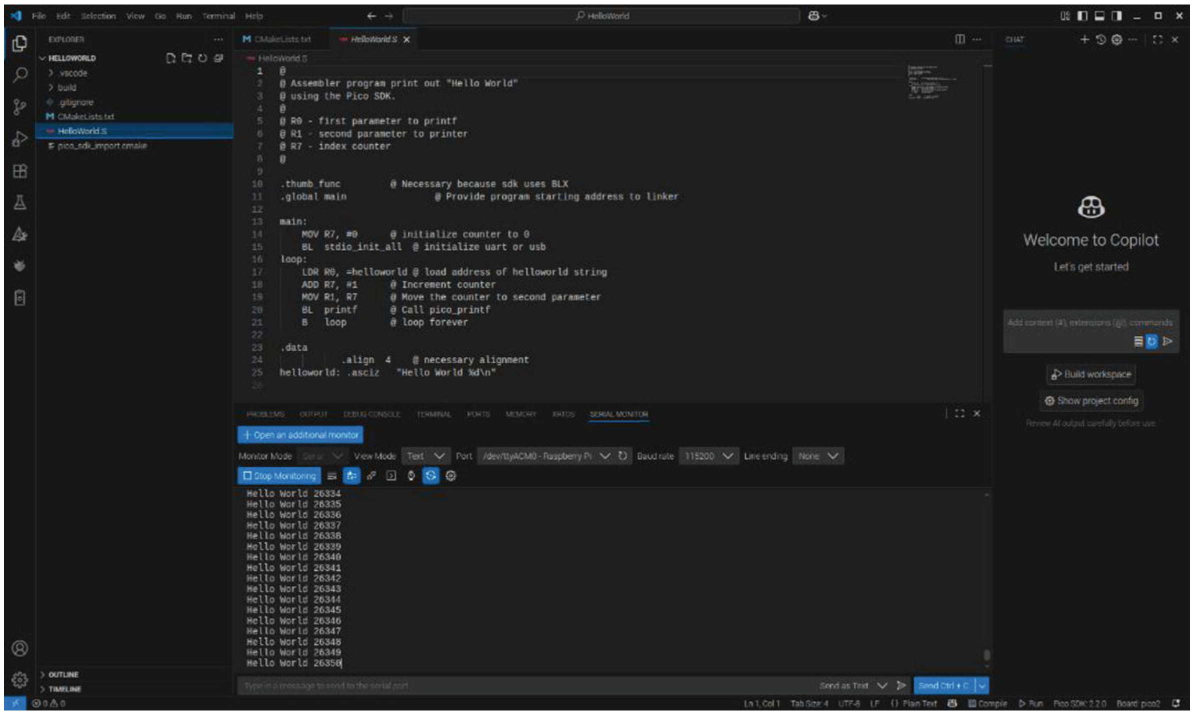
The complete **CMakeLists.txt** file is shown in Listing 2-2.

```
cmake_minimum_required(VERSION 3.13)

set(CMAKE_C_STANDARD 11)
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_EXPORT_COMPILE_COMMANDS ON)

# Initialise pico_sdk from installed location
# (note this can come from environment, CMake cache etc)

# == DO NOT EDIT THE FOLLOWING LINES for the Raspberry Pi
Pico VS Code Extension to work ==
if(WIN32)
    set(USERHOME $ENV{USERPROFILE})
else()
    set(USERHOME $ENV{HOME})
endif()
set(sdkVersion 2.2.0)
set(toolchainVersion 14_2_Rel1)
set(picotoolVersion 2.2.0)
set(picoVscode ${USERHOME}/.pico-sdk/cmake/pico-vscode.cmake)
if (EXISTS ${picoVscode})
    include(${picoVscode})
endif()
#
=============================================================
set(PICO_BOARD pico2 CACHE STRING "Board type")

# Pull in Raspberry Pi Pico SDK (must be before project)
include(pico_sdk_import.cmake)

project(HelloWorld C CXX ASM)

# Initialise the Raspberry Pi Pico SDK
pico_sdk_init()

# Add executable. Default name is the project name, version
0.1

add_executable(HelloWorld HelloWorld.S )

pico_set_program_name(HelloWorld "HelloWorld")
pico_set_program_version(HelloWorld "0.1")

# Modify the below lines to enable/disable output over
UART/USB
pico_enable_stdio_uart(HelloWorld 1)
pico_enable_stdio_usb(HelloWorld 0)

# Add the standard library to the build
target_link_libraries(HelloWorld
        pico_stdlib)
```

```
# Add the standard include files to the build
target_include_directories(HelloWorld PRIVATE
        ${CMAKE_CURRENT_LIST_DIR}
)

pico_add_extra_outputs(HelloWorld)
```

Listing 2-2 CMakeLists project definition file

With this work done, click the **Compile** button on the bottom status bar to compile the project. For best results disconnect the USB cable from the Pico-series and reconnect it while pressing the **BootSel** button. Then start the Serial Monitor in VS Code. Now click **Run** and the Hello World strings should be seen in the monitor pane as shown in Figure 2-5.



```
@
@ 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 helloworld 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"
```

Monitor Mode: Serial View Mode: Text Port: /dev/ttyACM0 - Raspberry Pi Baud rate: 115200 Line ending: None

```
Hello World 26334
Hello World 26335
Hello World 26336
Hello World 26337
Hello World 26338
Hello World 26339
Hello World 26340
Hello World 26341
Hello World 26342
Hello World 26343
Hello World 26344
Hello World 26345
Hello World 26346
Hello World 26347
Hello World 26348
Hello World 26349
Hello World 26350
```

Figure 2-5 VS Code running the HelloWorld program showing the results in the Serial Monitor pane

With the Pico-series C/C++ SDK

First, create a "pico" folder in $HOME and then create a "HelloWorld" folder in the $HOME/pico folder. Now copy the Assembly Language source file **HelloWorld.S** from Listing 2-1 to this folder. Next, copy **CMakeLists.txt** from Listing 2-2. All the files mentioned here will be placed in this folder.

The **CMakeLists.txt** file lists the source files, the libraries needed, and some configuration details for the SDK. This file will compile the **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, which mostly match the SDK requirements since the included parts of the SDK need to be built to be included in the program.

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

Copy **pico_sdk_import.cmake** from the SDK folder **pico-sdk/external** into the project folder. Finally, create a **build** folder using "mkdir build" or using the file explorer. The project folder should now look like the following:

```
drwxr-xr-x 5 smist08 smist08 4096 Aug 12 15:25 build
-rw-r--r-- 1 smist08 smist08 1564 Aug 12 13:49
CMakeLists.txt
-rw-r--r-- 1 smist08 smist08  664 Nov  5  2021 HelloWorld.S
-rw-r--r-- 1 smist08 smist08 6022 Aug  5 11:36
pico_sdk_import.cmake
```

The project is now ready to build. Open a terminal window and **cd** into the project folder's build folder. Type

```
cmake ..
```

Note It might be necessary to install cmake with "sudo apt install cmake."

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

```
make
```

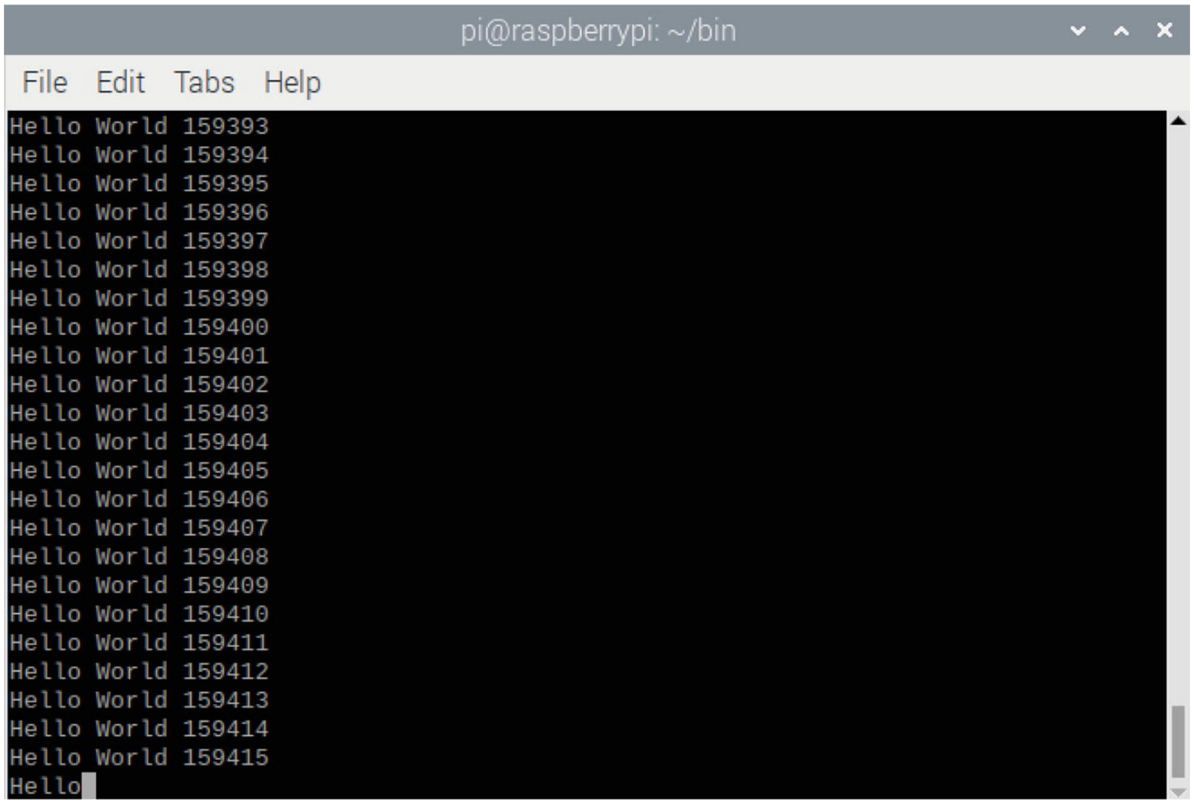
This command compiles the project. If all goes well, the build folder should now contain the following:

```
-rw-r--r-- 1 smist08 smist08  28562 Aug 12 15:24
CMakeCache.txt
drwxr-xr-x 6 smist08 smist08   4096 Aug 12 15:25 CMakeFiles
-rw-r--r-- 1 smist08 smist08   1837 Aug 12 15:24
cmake_install.cmake
-rw-r--r-- 1 smist08 smist08 362565 Aug 12 15:24
compile_commands.json
drwxr-xr-x 3 smist08 smist08   4096 Aug 12 15:24 generated
-rwxr-xr-x 1 smist08 smist08  15292 Aug 12 15:25
HelloWorld.bin
-rw-r--r-- 1 smist08 smist08 237755 Aug 12 15:25
HelloWorld.dis
-rwxr-xr-x 1 smist08 smist08 393212 Aug 12 15:25
HelloWorld.elf
```

```
-rw-r--r-- 1 smist08 smist08 385508 Aug 12 15:25
HelloWorld.elf.map
-rw-r--r-- 1 smist08 smist08  43081 Aug 12 15:25
HelloWorld.hex
-rw-r--r-- 1 smist08 smist08  31232 Aug 12 15:25
HelloWorld.uf2
-rw-r--r-- 1 smist08 smist08 108722 Aug 12 15:24 Makefile
-rw-r--r-- 1 smist08 smist08     60 Aug 12 15:24
pico_flash_region.ld
drwxr-xr-x 6 smist08 smist08   4096 Aug 12 15:24 pico-sdk
```

HelloWorld.uf2 is the compiled program. It can be run by powering off the Pico-series 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 HelloWorld.uf2 can be copied onto that drive. As soon as this is done, the Pico will reboot and run the program.

The output can be viewed using minicom, if the batch files recommended in Chapter 1 were created. Then run m-usb assuming that the Debug Probe is being used. When this is done, something like the screenshot in Figure 2-6 should be observed.



```
Hello World 159393
Hello World 159394
Hello World 159395
Hello World 159396
Hello World 159397
Hello World 159398
Hello World 159399
Hello World 159400
Hello World 159401
Hello World 159402
Hello World 159403
Hello World 159404
Hello World 159405
Hello World 159406
Hello World 159407
Hello World 159408
Hello World 159409
Hello World 159410
Hello World 159411
Hello World 159412
Hello World 159413
Hello World 159414
Hello World 159415
Hello
```

Figure 2-6 The output from the minicom program for Hello World

Now that the program is running, the contents of **HelloWorld.S** are examined.

Our First Assembly Language File

This file is organized into four sections: the header comments, the function definition, the Assembly Language code, and the program data. Each of these sections will be examined in detail.

About the Starting Comment

The program starts with a comment that states what it does. It documents the registers used since keeping track of which registers are doing what becomes important as our programs get bigger.

- An "@" character is the comment character, and 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 is going on. Otherwise, returning to the program after a couple of weeks will result in having 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, the starting point of the program is specified.

- This is defined as a global symbol called main that the Pico-series runtime will call to execute the program. All programs will contain this somewhere.
- This must be defined as a thumb_func, due to the way the SDK calls the function. What this means is explained in Chapter 7. Cortex M-series CPUs don't support any other type of function, but this is still required. If omitted, a hardware fault will result when the program is run.
- The program can consist of multiple .S files, but only one can contain main.

Assembly Instructions

Five different Assembly Language instructions are used in this example:

1. **MOV**, which moves data into a register. First, an immediate operand is used, which starts with the "#" sign. So "MOV R7, #0" means move the number 0 into **R7**. In this case the 0 is part of the instruction and not stored elsewhere in memory. Secondly, "MOV R1, R7" 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. Two functions are called: **stdio_init_all** to initialize communications back to the Raspberry Pi 5 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.
   LDR. which is used to both load memory addresses and load the contents for

3. memory. In this case "LDR R0, =helloworld" loads register **R0** with the address of the string 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 is the **.data** statement, which indicates the following instructions are located in the data section of the program.

- First, there is 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 non-aligned 32-bit quantity).
- Next is the label "helloworld" followed by an **.asciz** statement and then the string to print.
- The **.asciz** statement tells the Assembler to put the string in the data section, and then it can be accessed via the label as done 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. How text is represented as numbers will be discussed later; the encoding scheme here is called ASCII.
- The last "\n" character is how a new line character is represented.

These are the individual instructions. Now how they work together is discussed.

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 the application program. The expectation is that the program will be run shortly after the Pico-series powers on and then terminated when it is powered off. This is why an infinite loop was created that runs forever, which is typical of most microcontroller programs.

If the program terminated after printing "Hello World", the CPU would halt until the microcontroller is powered off and on again. Chances are the printing of "Hello World" would be missed because it would happen before minicom is started. A counter was added as a simple example and so that when minicom is run it is clear that something is 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 was configured in the CMakeLists.txt file. For the Raspberry Pi Debug Probe, this should be UART.

The call to **printf** is an alias to **pico_printf,** which is an implementation of the C runtime's **printf** but contained in the Pico-series SDK for anyone to use. Assembly Language programmers can call anything there as long as they know the protocol to do so.

Why keep the 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 is put there. If **printf** uses **R7,** then it must save the value and restore it before returning. The register usage protocol will be studied 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, since it converts the binary 32-bit quantity into a human-readable number. Hopefully, if familiar with C programming, then this is all basic and familiar.

Reverse Engineering the Program

How each Assembly Language instruction is compiled into a 16-bit number was touched on quickly. The Assembler created the binary version of HelloWorld, and it provides a file to show what it did. Specifically, look at the **HelloWorld.dis** file that was generated in the build folder. This file contains everything that is combined to create the program. This includes the code to initialize the RP2040 or RP2350 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 the code and data sections from Listing 2-1.

```
10000234 <main>:
10000234:       2700            movs    r7, #0
10000236:       f002 fe8d       bl      10002f54
<stdio_init_all>

1000023a <loop>:
1000023a:       4803            ldr     r0, [pc, #12]       @
(10000248 <loop+0xe>)
1000023c:       3701            adds    r7, #1
1000023e:       1c39            adds    r1, r7, #0
```

```
10000240:      f002 ff50       bl     100030e4
<__wrap_printf>
10000244:      e7f9            b.n    1000023a <loop>
10000246:      0000            .short 0x0000
10000248:      200005b0        .word  0x200005b0
...
200005b0 <helloworld>:
200005b0:      6c6c6548        .word  0x6c6c6548
200005b4:      6f57206f        .word  0x6f57206f
200005b8:      20646c72        .word  0x20646c72
200005bc:      000a6425        .word  0x000a6425
```

Listing 2-3 Disassembly of Hello World

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 an "@" comment character.

Some points to notice from this listing:

- Most of the instructions compile to 16-bit quantities except for the **BL** statements, which are 32 bits. Practically speaking if the M-series CPU insisted on making **BL** statements 16 bits, then the jumps would be too small to be useful, and the only alternative would be to build the address in a register and then jump to it indirectly, which would take several statements. This way functions can be called efficiently 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 features from the A-series processor are present, even though these can't be changed on the M-series CPUs.
- 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. The Assembler will use the most efficient version possible.
- 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 is wanted. With only 16 bits, bits can't be wasted 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, so the programmer doesn't need to remember all these tricks that go on under the hood.

Look at the **LDR** instruction. It changed from

```
ldr    R0, =helloworld
```

to

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

This is the Assembler helping with the ARM processor's mechanism of addressing memory with one instruction. It allows a symbolic address to be specified, namely, "helloworld," and translate that into an offset from the program counter.

Note [pc, #12] points to a bit of memory that holds 20000180, which is the actual address of the "Hello World" string. The Assembler inserted this, and it will be covered in detail in Chapter 6.

The 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 M-series processors. 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 this code, it contains the SDK runtime code to initialize the RP2040/RP2350, set up the environment, and then run the 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/520kb of memory available to the RP2040/RP2350. 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 Pico-series after it powers up. A bit of code in the Pico-series firmware loads the code into memory and then passes execution to it, and away it goes.

Summary

This chapter introduced the ARM Cortex M-series processor and Assembly Language programming along with why to use Assembly Language. Some of the tools that will be used throughout the book were covered. How computers represent positive integers was explained. How the ARM CPU represents Assembly Language instructions was studied along with the registers it contains for processing data. The RP2040/RP2350's memory was introduced. The GNU Assembler was introduced, which will assist in writing Assembly Language programs. A simple complete program to print "Hello World" was written, and its output was viewed in VS Code or minicom on the Raspberry Pi. In Chapter 3, more details on the tools used to build and debug programs will be studied.

Exercises

1. Convert the decimal number 1234 to both binary and hexadecimal.
2. Download the source code for this book from GitHub and compile the HelloWorld program on a Raspberry Pi. Next, run it on a Pico-series board and observe the output in minicom or VS Code.

3. Compare the size of the uf2 file when setting the various output options between none, UART, and USB. Remember to delete the build folder whenever changing the CMakeLists.txt file. Which one is the better option as the program size approaches 264kb?
4. Decode a couple of the binary format of the instructions in Listing 2-3 to figure out the operand and where the registers are specified.
5. Change the string that is printed. Try printing the number in hexadecimal.
6. Rather than count up, change the program to count down subtracting 1 rather than adding 1 in the loop.

S. Smith, *RP2040 Assembly Language Programming*, Maker Innovations Series
https://doi.org/10.1007/979-8-8688-2202-5_3

3. How to Build and Debug Programs

Stephen Smith[1]
(1) Gibsons, BC, Canada

In this chapter, the build tools employed for program development are explored in greater detail. The Pico-series C/C++ SDK and the Visual Studio Code extension streamline much of the process of building programs, yet gaining insight into the operations beneath these high-level tools can be highly beneficial. Following this, attention turns to the GNU debugger (**gdb**), which enables single stepping through programs and examining registers and memory during execution.

CMake

CMake is an open source, build automation tool that is cross-platform and compiler independent. The goal of using **CMake** in the Pico-series 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. Clicking the **Compile** button in VS Code results in CMake being run. With **CMake** the project is built from the **CMakeLists.txt** file, and the details of how to run the GNU Assembler are automated. To fully cover **CMake** requires a full book in itself, so only what is needed for Assembly Language programming is covered.

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 compiling for the correct ARM Cortex M-series processor, and lets **CMake** know where all the SDK files are located. The goal is to specify the target executable name and list the 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 the project; instead, it creates a **makefile** for the GNU Make tool, which is covered 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 that **CMake** created. Now a selection of the contents from the **CMakeLists.txt** file from Listing 2-2 is examined.

```
cmake_minimum_required(VERSION 3.13)
```

The above 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.

```
set(CMAKE_C_STANDARD 11)
set(CMAKE_CXX_STANDARD 17)
set(sdkVersion 2.2.0)
set(toolchainVersion 14_2_Rel1)
set(picotoolVersion 2.2.0)
```

These above 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. Then the version of the Pico-series SDK, the version of the GNU toolchain, and the version of the picotool are specified; the picotool is responsible with interacting with the Pico-series board after **bootsel** is pressed to perform tasks like downloading new programs to the board.

```
set(PICO_BOARD pico2 CACHE STRING "Board type")
```

Setting the PICO_BOARD variable is crucial. This value ends up in the resulting executable, and it won't run unless this value matches the value expected by the board. The default is the Pico 1 with the RP2040 chip, so this is crucial for any Pico 2 RP2350-type board.

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

The include statement includes the code from the specified file into the file and executes it. This file was copied into the same place as the **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 above line defines the project name as HelloWorld and that C, C++, and Assembly Language will be used. Even though the project didn't include any C or C++ files, many such files were included from the SDK.

```
pico_sdk_init()
```

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

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

The above statement is where to add source files. A couple of extra files were added for demonstration purposes.

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 that is needed is to add files here and CMake will take care of the rest.

```
pico_set_program_name(HelloWorld "HelloWorld")
pico_set_program_version(HelloWorld "0.1")
```

These two lines set the program name and version, which are embedded in the resulting executable file.

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

The above macros are defined in the Pico's SDK. We set them to control where the output from **printf** statements go. 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.

```
target_link_libraries(HelloWorld pico_stdlib)
```

The above statement specifies the libraries to use. The library needed so far is pico_stdlib, but other libraries can be added as needed.

```
target_include_directories(HelloWorld PRIVATE
        ${CMAKE_CURRENT_LIST_DIR}
)
```

The above call sets up where to look for include directories. If unchanged this call includes all the various source files in the SDK. If the project has the source code spread over multiple folders, then these can be added separating them by spaces.

```
pico_add_extra_outputs(HelloWorld)
```

If the above line is left out, the build works and an .elf file is produced, which is an executable file for Linux; however, this isn't always what is wanted. 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 Pico-series' flash storage. It also generates useful files like the .dis file (disassembly file).

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. **CMake** writes all the dependency scripts, so the details of **makefiles** won't be covered here. However, Make needs to be run when using the Pico-series SDK after CMake is finished.

To build everything, ignoring the file data/times, use

```
make -B
```

Print Statements

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

- Functions are allowed to use registers **R0–R3** without saving them. If any of these four registers were used, 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 code relying on the **CPSR**.
- Each time seeing something new is required, adding a **printf** call is needed, adding code to set registers and call the function. Then everything needs to be recompiled, the .uf2 file copied to the Pico-series board, and the output observed.
- There is only 264kb/520kb of memory on the RP2040/RP2350, and creating a lot of strings to print things can use a substantial amount of this precious resource.
- Even though the SDK is lightweight, it still takes memory and adds processing time to the program, perhaps disrupting time-sensitive tasks.
- Adding and removing source code for the **printf** statements could result in bugs, for example, if a mistake is made and one extra instruction is deleted.
- There may be surprising side effects from executing **printf** that disrupt the program.

Some of these problems can be alleviated by using the GNU Assembler's macro feature. How to do this will be looked at in Chapter 7*. I*n addition, **printf** is a useful function, but to address these limitations, what is really needed 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 the Assembly Language programming, but also it is a great tool to use with 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 either by the VS Code extension or the pico_setup.sh script. This section assumes using the Raspberry Pi

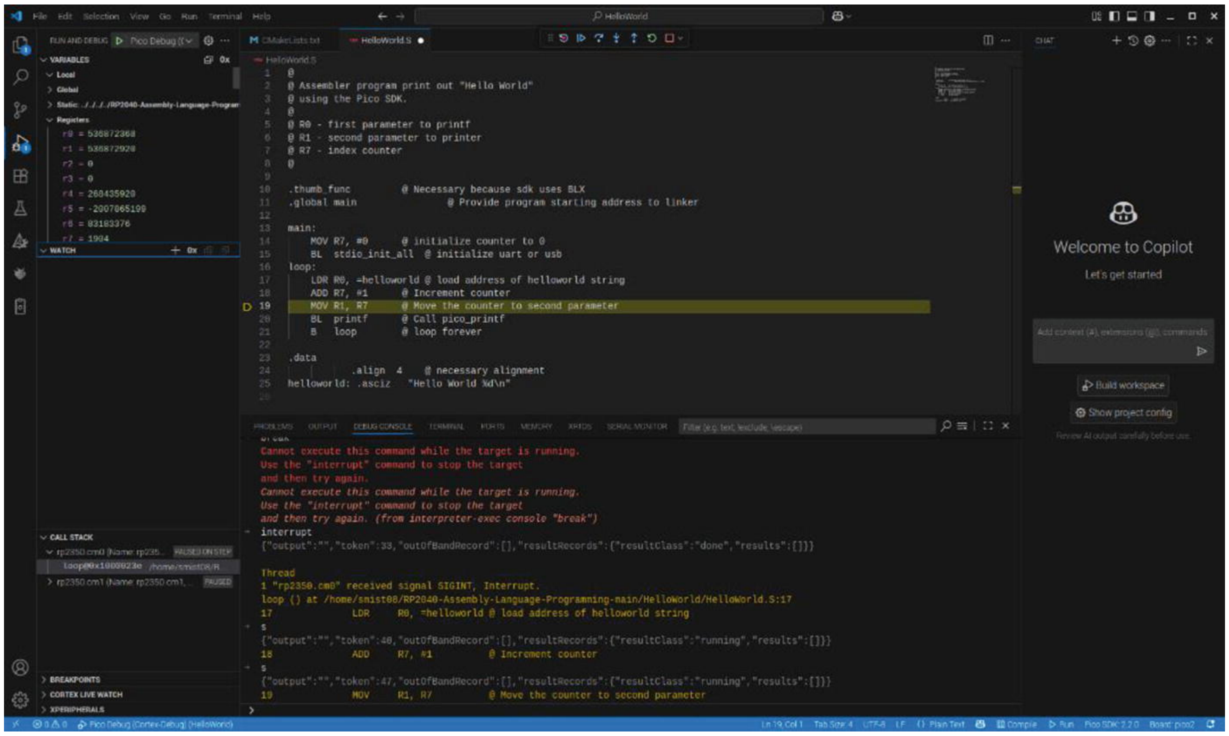
Debug Probe.

Using the VS Code Extension

All debugging can be done inside Visual Studio Code. This provides a nice visual environment for debugging.

> **Note** Make sure to rename the **.gdbinit** file given in Chapter 1; this is for debugging outside of VS Code, and its presence will cause **gdb** to not start inside VS Code.

To start the debugging from VS Code, simply select "Start Debugging ..." from the Run menu. This launches gdb and creates a breakpoint at main as a starting point. This view provides a number of useful panes such as a view of the current values of the registers. Figure 3-1 shows a common **gdb** session for the HelloWorld program.



```
r0 = 536872368
r1 = 536872928
r2 = 0
r3 = 0
r4 = 268435920
r5 = -2007065199
r6 = 83183376
r7 = 1904
```

```
@
@ 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 helloworld 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"
```

```
Cannot execute this command while the target is running.
Use the "interrupt" command to stop the target
and then try again.
Cannot execute this command while the target is running.
Use the "interrupt" command to stop the target
and then try again. (from interpreter-exec console "break")
interrupt
{"output":"","token":33,"outOfBandRecord":[],"resultRecords":{"resultClass":"done","results":[]}}

Thread
1 "rp2350.cm0" received signal SIGINT, Interrupt.
loop () at /home/smist08/RP2040-Assembly-Language-Programming-main/HelloWorld/HelloWorld.S:17
17              LDR     R0, =helloworld @ load address of helloworld string
s
{"output":"","token":40,"outOfBandRecord":[],"resultRecords":{"resultClass":"running","results":[]}}
18              ADD     R7, #1          @ Increment counter
s
{"output":"","token":47,"outOfBandRecord":[],"resultRecords":{"resultClass":"running","results":[]}}
19              MOV     R1, R7          @ Move the counter to second parameter
```

Figure 3-1 Running GDB inside Visual Studio Code

gdb commands are entered at the bottom of the Debug Console pane. Take care that after each command focus is set to the code window rather than staying in the debug console.

Next, how to get started without using VS Code will be looked at. Then a selection of **gdb** commands will be looked at, which will apply to both environments.

Preparing to Debug

VS Code handles setting up to debug behind the scenes, but when using the raw Pico-series SDK, there is a bit of preparation required. The GNU debugger (**GDB**) can debug programs as it is, but this isn't the most convenient way to go. In the HelloWorld program there is the label **helloworld**. If the program is debugged 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 they can be used in the debugger. This makes the program executable a bit larger. The Assembler command-line arguments don't need to be known; instead, CMake is provided with a command-line argument to specify a debug build.

Often, debug mode is set while developing the program and then turned off 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 non-debug modes.

Generally it isn't a good idea to leave debug information in programs 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 the program is open source, then this isn't important as anyone can look at the source code and build the program with any options desired. There are several cases where hackers caused mischief because the program still had debug 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 the program, invoke **CMake** setting the CMAKE_BUILD_TYPE to Debug. To ensure everything is generated properly, delete and recreate the build folder first:

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

Note The **cmaked** script from Chapter 1 could have been used to save some typing.

Now everything is set up for debugging.

Beginning GDB

Before starting the debugger, the **openocd** server needs to run:

```
sudo openocd -f interface/cmsis-dap.cfg -f
     target/rp2350.cfg -c "adapter speed 5000"
```

Or use the ocdg script created in Chapter 1.

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

```
gdb HelloWorld.elf
```

This yields the abbreviated output:

```
$ gdb HelloWorld.elf
GNU gdb (Debian 13.1-3) 13.1
Copyright (C) 2023 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
...
warning: No executable has been specified and
target does not support
determining executable automatically.  Try using
the "file" command.
```

```
warning: multi-threaded target stopped without
sending a thread-id, using first non-exited thread
0x1000023e in ?? ()
Reading symbols from HelloWorld.elf...
(gdb)
```

The warning is a side effect of programming a microcontroller and there is no operating system. It means the program isn't ready to run yet; one more command needs to be entered to load it first.

Note If a **.gdbinit** file as indicated in Chapter 1 isn't present, then enter the command "target remote localhost:3333" at this point to connect to the Pico-series board.

- gdb is a command-line program.
- (gdb) is the command prompt where commands are typed.
- Hit Tab for command completion. Enter the first letter or two of a command as a shortcut.

First, the program needs to be loaded; type

```
load
```

(or lo for short). This can be done repeatedly, so in another window, changes to the program can be made and recompiled, and then load it again. This way the **gdb** environment doesn't need to be restarted for each program change, and any commands entered like setting breakpoints are still in effect. 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).

If minicom is run to configure to read the Debug Probe, the "Hello World" strings will be seen going by. The program will run forever, but can be stopped by typing control-c.

After terminating the program, it will either be inside **HelloWorld.S** code or inside one of the Pico-series SDK's routines.

To stop at the start of HelloWorld, set a breakpoint to stop in the 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.

Thread 1 "rp2350.cm0" hit Breakpoint 1, main ()
    at /home/smist08/RP2040/Chapter
2/HelloWorld.S:14
14 MOV R7, #0 @ initialize counter to 0
```

As far as **gdb** is concerned, the whole .elf file is the program, including the SDK code to initialize the Pico-series. Since the entire SDK is provided as source code, anything that is described here for debugging code works equally well for the SDK code. The provision is that the SDK code needs to do initial setup on the RP2040/RP2350 before a breakpoint can stop the CPU.

To list the program, type

```
list
```

(or l).

This lists ten lines. Type

```
l
```

for the next ten lines. Type

```
list 1,1000
```

to list the entire program.

The list gives the source code for the program, including comments. This is a handy way to find line numbers for other commands. If the raw machine code needs to be examined, then **gdb** can disassemble the program with

```
disassemble main
```

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

The program can be executed one instruction at a time with the step command (or s). To see the values of the registers, use the info registers (or i r) command:

```
Thread 1 "rp2350.cm0" hit Breakpoint 1, main ()
    at /home/smist08/RP2040/Chapter
2/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             0xe000e434          -536812492
r1             0x10000235          268436021
r2             0x80808080          -2139062144
r3             0x1000318c          268448140
r4             0x100001d0          268435920
r5             0x88526891          -2007865199
r6             0x4f54710           83183376
r7             0x0                 0
r8             0x43280035          1126694965
r9             0x0                 0
r10            0x10000000          268435456
r11            0x62707361          1651536737
r12            0x4a6dc800          1248708608
sp             0x20082000          0x20082000
lr             0x1000018f          268435855
```

```
pc             0x10000236          0x10000236
<main+2>
xpsr           0x69000000          1761607680
fpscr          0x0                 0
msp            0x20082000          0x20082000
psp            0x0                 0x0
msp_ns         0x0                 0x0
psp_ns         0xfffffffc          0xfffffffc
msp_s          0x20082000          0x20082000
psp_s          0x0                 0x0
primask        0x0                 0
basepri        0x0                 0
faultmask      0x0                 0
control        0x0                 0
msplim_s       0x0                 0x0
psplim_s       0x0                 0x0
msplim_ns      0x0                 0x0
psplim_ns      0x0                 0x0
primask_s      0x0                 0
basepri_s      0x0                 0
faultmask_s    0x0                 0
control_s      0x0                 0
primask_ns     0x0                 0
basepri_ns     0x0                 0
faultmask_ns   0x0                 0
control_ns     0x0                 0
```

R7 was set to 0 as expected. Continue single stepping or enter continue (or c) to continue to the next breakpoint if there is one. As many breakpoints as required can be set. These can be seen 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
4       breakpoint     keep y   0x10000234
/home/smist08/RP2040/Chapter 2/HelloWorld.S:14
```

```
(gdb) delete 4
(gdb) i b
No breakpoints or watchpoints.
(gdb)
```

Memory hasn’t been studied yet, but **gdb** has good mechanisms to display memory in different formats. The main command is 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 0x10000234:

```
(gdb) x /4ubft main
0x10000234 <main>: 00000000 00100111 00000010
11110000
(gdb) x /4ubfi main
=> 0x10000234 <main>: movs r7, #0
   0x10000236 <main+2>: bl 0x10002d10
<stdio_init_all>
   0x1000023a <loop>: ldr r0, [pc, #12] @
(0x10000248 <loop+14>)
```

```
   0x1000023c <loop+2>: adds r7, #1
(gdb) x /4ubfx main
0x10000234 <main>: 0x00 0x27 0x02 0xf0
(gdb) x /4ubfd main
0x10000234 <main>: 0 39 2 -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 new things are learned, the knowledge of **gdb** will be enhanced. It is a powerful tool to help develop programs. Assembly Language programs are complex and subtle, and **gdb** is great at showing 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 what each instruction is doing is understood.

Even if there isn’t a known bug, many programmers like to single step through the code to look for problems and to convince themselves that the code is correct. Often two programmers do this together as part of the pair programming agile methodology.

Summary

In this chapter, the **CMake** program was introduced that will be used to build programs. This is a powerful tool used to generate all the rules for the various compilers and linkers needed. Then the GNU debugger was introduced that will allow the troubleshooting of programs. Unfortunately, programs have bugs, and a way is needed to single step through them and examine all the registers and memory as through this process. **GDB** is a technical tool, but it's indispensable in figuring out what programs are doing.

In Chapter 4, how to load data into the CPU registers and performing basic arithmetic will be studied. How negative numbers are represented is covered along with learning new techniques for manipulating binary bits.

Exercises

1. Step through the "Hello World" program from Chapter 2, to ensure complete understanding of the changes each instruction makes to the registers. Ensure the output of the print statements can be seen.
2. Experiment with the various **gdb** commands to ensure familiarity with their various options.
3. Why does **CMake** generate a **makefile** that is used to build a program rather than building it itself?

S. Smith, *RP2040 Assembly Language Programming*, Maker Innovations Series
https://doi.org/10.1007/979-8-8688-2202-5_4

4. How to Load and Add

Stephen Smith[1]
(1) Gibsons, BC, Canada

This chapter introduces the **MOV**, **ADD**, and **SUB** instructions, first by gradually providing a foundation for understanding the functionality of the commands, particularly in how parameters (operands) are handled. In subsequent chapters, the rest of the ARM instruction set is covered at a faster pace. Before delving into the specifics of **MOV**, **ADD**, and **SUB** instructions, topics such as the representation of negative numbers, as well as the concepts of shifting and rotating bits, will be addressed.

About Negative Numbers

In the previous chapter, the representation of positive integers as binary numbers, known as unsigned integers, was discussed. However, a question arises: how are negative numbers represented? One intuitive approach might suggest designating a single bit to indicate whether a number is positive or negative. While straightforward, this method introduces additional complexity for the CPU, as processing would require checking the sign bit and then determining the appropriate arithmetic operation and ordering.

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.

Consider a 1-byte hexadecimal number like 01. If 0xFF is added

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

(all binary ones) the result is 0x100. However, since these are 1-byte numbers, then the 1 is overflow and the result is zero:

```
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 in the realm of 1-byte integers. The two’s complement form for any number can be obtained by taking

```
2^N - number
```

In the 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 two’s complement is to change all the 1s to 0s and all the 0s to 1s and then add 1. Doing this to 1 results in

```
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 computers represent negative integers this way? As it turns out, addition is simple for the computer to execute. There are no special cases; if the overflow is discarded, 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 the size is limited to 1 byte or 8 bits, the leading 1 overflows, and the result is 2.

About the Raspberry Pi OS Calculator

Fortunately, computers provide good tools to do the conversions and arithmetic for us, but when signed numbers are seen in memory, these need to be recognized for what they are. The Raspberry Pi OS calculator calculates two's complement; 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.

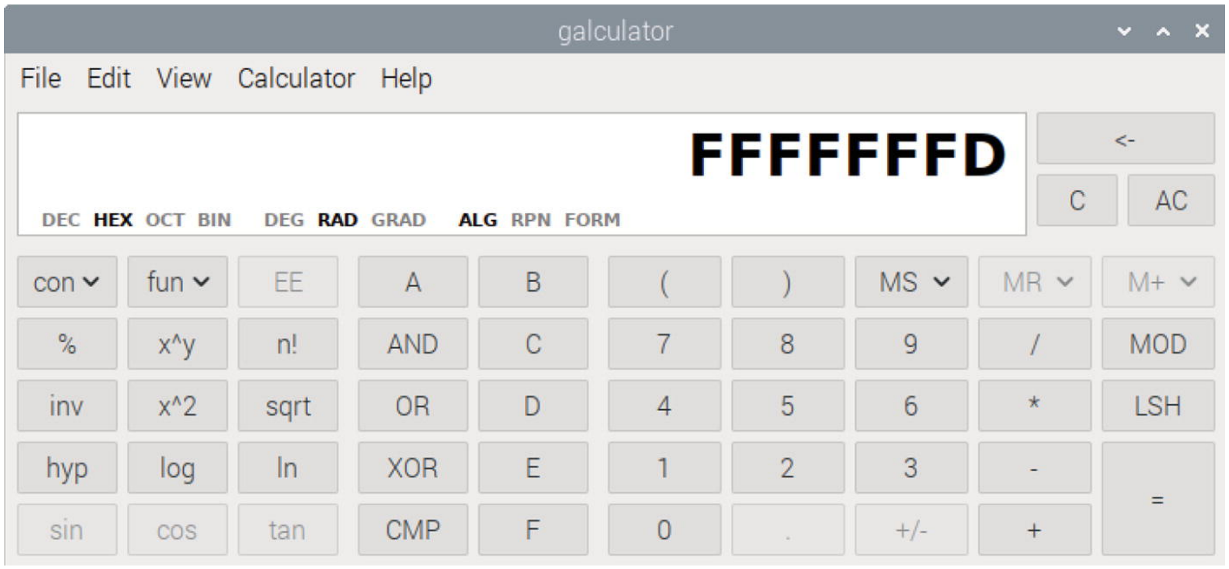

galculator
File Edit View Calculator Help
FFFFFFFD
DEC HEX OCT BIN DEG RAD GRAD ALG RPN FORM
<-
C AC
con fun EE A B ( ) MS MR M+
% x^y n! AND C 7 8 9 / MOD
inv x^2 sqrt OR D 4 5 6 * LSH
hyp log ln XOR E 1 2 3 - =
sin cos tan CMP F 0 . +/- +

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

About One's Complement

Change all the 1s to 0s and vice versa; then this is called **one's complement**, like two's complement but without adding 1. There are uses for the **one's complement** form, and these will be encountered in later chapters.

Big- versus Little-Endian

When examining 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 the Little-Endian format and store 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.

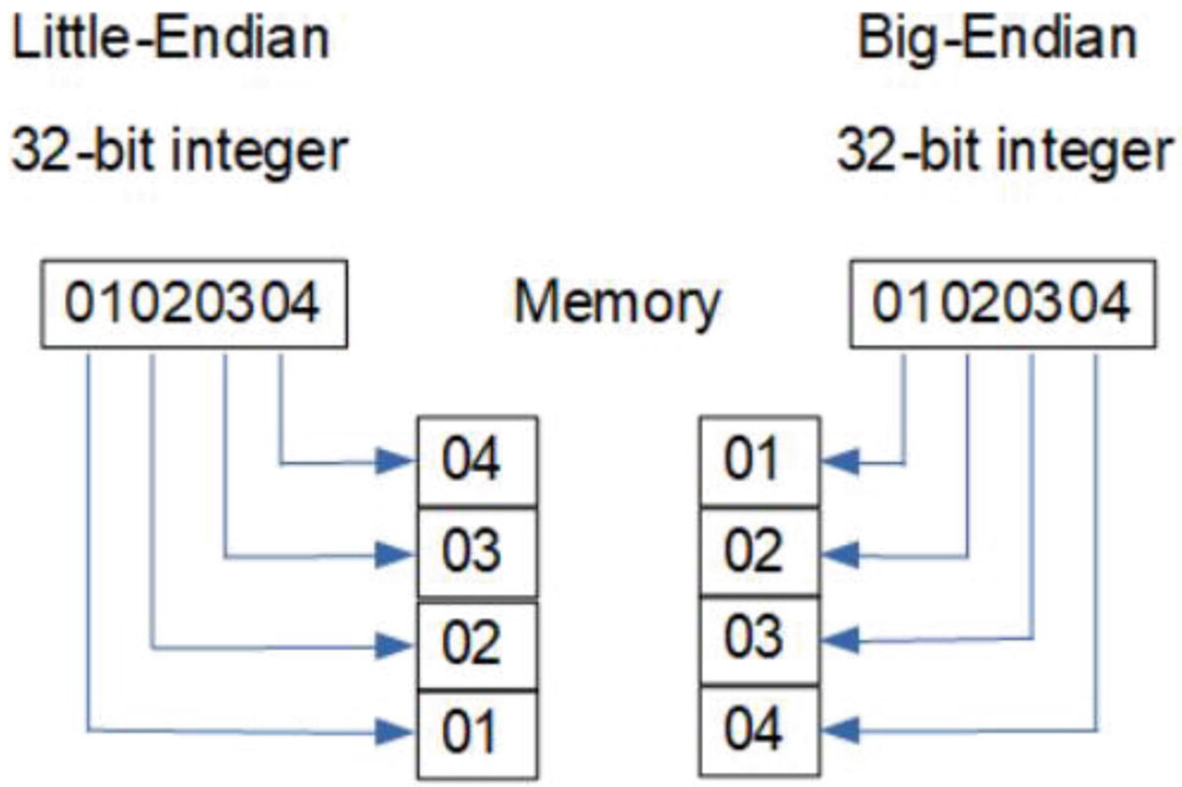


Figure 4-2 How integers are stored in memory in Little- versus Big-Endian formats

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 Pico-series 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. To convert a 4-byte integer to a 1-byte integer, load the first byte, assuming the integer is in the range of 0–255 and the other 3 bytes are zero. For example, if memory contains the 4 bytes or word for 1, in Little-Endian, the memory contains

```
01 00 00 00
```

If a 1-byte representation of this number is needed, take the first byte; for the 16-bit representation, take the first 2 bytes. The key point is that the memory address used is the same in all cases, saving an instruction cycle to adjust it.

Cons of Little-Endian

Even though the Pico-series 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/RP2350 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

There are sixteen 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 to get

```
0011 0000
```

which is

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

Shifting 0x30 right by 4 bits undoes this showing it is equivalent to dividing by 2^4.

About the Carry Flag

In the **CPSR**, there is a bit for **carry**. This is normally used to perform addition on larger numbers. When adding two 32-bit numbers and the result is larger than 32 bits, the carry flag is set. How to use this in the case of addition will be looked at in detail later in this chapter. When shifting and rotating, it turns out to be useful to include the carry flag. This allows doing conditional logic based on the last bit shifted out of the register.

Basics of Shifting and Rotating

There are five cases to cover, as follows:

- Logical Shift Left
- Logical Shift Right
- Arithmetic Shift Right
- Rotate Right
- Rotate Right Extend

Logical Shift Left

This is quite straightforward, as the bits are shifted 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

As the bits are shifted right, zeros come in from the left, and the last bit shifted out on the left 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 the sign bit needs to be preserved, instead use Arithmetic Shift Right. 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 when 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 in the instruction.

How to Use MOV

This section covers the two forms of the **MOV** instruction:

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

Move Immediate

The first case is move immediate, which puts a small number into a register. Here the immediate value can be any 8-bit quantity, and it will be placed in the lower 8 bits of the specified register. This form of the **MOV** instruction is as simple as it gets and will be used 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

The second case is 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** and another that moves between any registers but doesn't set the **CPSR**. This is one of the few instructions that allows access to the high registers **R8–R15**.

Note Remember that **R12–R15** are special and changing these will have side effects. **R12** is the intra-procedure call scratch register (**IP**), **R13** is the stack pointer (**SP**), **R14** is the link register (**LR**), and **R15** is the program counter (**PC**). Moving a value to **R15** will cause execution to jump to that location. How to properly use these registers will be studied in later chapters, so avoid them for now.

Here are some examples:

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

Now that small 8-bit values can be placed in registers, it is time to do some arithmetic.

ADD/ADC

Start with addition. The various forms of the addition instruction are

- ADD Rd, Rn, #imm3
- ADD Rd, Rd, #imm8
- ADD Rd, Rm, Rn
- ADD Rd, Rd, Rm
- ADD SP, SP, #imm7
- ADD Rd, SP, #imm8
- 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 as follows:

- 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. The function of these is covered 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 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 uses the carry flag from the **CPSR**.

Think back on how to add numbers:

```
 17
+78
 95
```

1. First, add 7 + 8 and get 15.

2. We put 5 in the sum and carry the 1 to the tens column.

3. Now add 1 + 7 + the carry from the ones column, so 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 it can be included in the sum of the next part.

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

The ARM processor adds 32 bits at a time, so the carry flag is only needed when dealing with numbers where the sum is larger than will fit into 32 bits. A common application is to use the carry flag to easily add 64-bit or larger numbers.

The carry flag is a bit in the **CPSR**; the **CPSR** will be looked at 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 the
desired register
```

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 those will be looked at later. The second instruction, **ADC**, adds the higher-order words, plus the carry flag.

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

The nice thing here is that although in 32-bit mode, 64-bit addition can be performed in only two clock cycles (three if the **MOV** is counted).

SUB/SBC

Subtraction is the inverse of addition. There are a number of forms of this:

- SUB Rd, Rn, Rm
- SUB Rd, Rn, #imm3
- SUB Rd, Rd, #imm8
- SBC Rd, Rd, Rn
- SUB SP, SP, #imm7
- NEG Rd, Rn

The operands are the same as those for addition, only now 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**). Here are 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 have **Rd** as the second operand can only operate in place (the first and second operands must be the same, and thus one can be omitted).

Here are 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
```

Quite a few instructions have been introduced in this chapter. Now on to combining a few of them to load a 32-bit register.

Loading All 32 Bits of a Register

So far, how to load 8 bits with an immediate operation has been seen; but, with **MOV** combined with shifting and adding, all the bits can be loaded, for example, to load **R0** with the value 0x12345678. The approach will be to do it 8 bits at a time. 8 bits will be loaded, shifted into position, and then added to the result. Listing 4-1 contains the code for this.

```
@ 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
```

Listing 4-1 Loading all 32 bits of a register

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, how to load registers from memory will be studied, which is less code, but there will be cases later where tricks like this result in quick ways to load registers (especially if there are zeroes in the middle). Next is an example containing all these instructions.

MOV/ADD/Shift Example

If the various code snippets in this chapter including the 32-bit register loading and 64-bit addition are combined, Listing 4-2 results. 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. The instructions are for using the Pico-series SDK, but the code could easily be put into a VS Code project.

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

```
@
@ 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
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 4-2 Examples of the MOV, ADD, and shift instructions along with 64-bit addition

Listing 4-3 contains the **CMakeLists.txt** file needed to build this sample. Be sure to change the PICO_BOARD value to the precise Pico-series board being used. Remember to copy **pico_sdk_import.cmake** to the project folder.

```
cmake_minimum_required(VERSION 3.13)
set(PICO_BOARD pico2 CACHE STRING "Board type")
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)
```

Listing 4-3 The CMakeLists.txt file for our sample

After building the program, have a look at **MovAddSub.dis**. The program consists of forty-seven 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 value set, which in this case isn't wanted. Registers **R12–R14** are used when functions are called, and if these are changed, the call to **printf** won't work. How to change **R15** is covered in the next chapter. How to use **R12–R14** is covered in Chapter 7.

Summary

This chapter explored how negative integers are represented in computers, followed by a discussion of Big- and Little-Endian byte ordering. The concept of shifting and rotating bits within a register was then introduced.

The next section provided a detailed examination of the **MOV** instruction, which facilitates transferring data between CPU registers or loading constants directly into a register.

Coverage included the **ADD** and **ADC** instructions, along with methods for adding both 32- and 64-bit numbers. A brief introduction to the **SUB** and **SBC** instructions was given. The discussion concluded with an overview of various shift and rotation instructions.

The instructions were combined to load all 32 bits of a register and then integrated into an example program to add two 64-bit integers.

In Chapter 5, conditionally executing code and branching and looping are covered, which are the core building blocks of programming logic.

Exercises

1. Compute the 8-bit two’s complement for –79 and –23.
2. What are the negative decimal numbers represented by the bytes 0xF2 and 0x83?
3. Manually write out the bytes in the Little-Endian representation of 0x12345678.
4. Manually write out the bytes for 0x23 shifted left by 3 bits.
5. Manually write out the bytes for 0x4300 right shifted by 5 bits.
6. Code a program to add two 96-bit numbers. Managing the limited number of registers will be a problem to be solved.
7. Code a program that performs 64-bit subtraction. Make sure that the way it sets and interprets the carry flag is understood. Use it to reverse the operations from the 64-bit addition in Listing 4-2.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
S. Smith, *RP2040 Assembly Language Programming*, Maker Innovations Series
https://doi.org/10.1007/979-8-8688-2202-5_5

5. How to Control Program Flow

Stephen Smith[1]
(1) Gibsons, BC, Canada

A handful of Assembly Language instructions are now familiar, allowing for linear execution, one after another. Programs can be built and debugged with these foundations. This chapter introduces more engaging program flow through conditional logic—such as **if/then/else** statements in high-level languages—and loops, including **for** and **while** constructs. With these instructions, the basics of coding logical program structures are established.

Unconditional Branch

The simplest branch instruction is

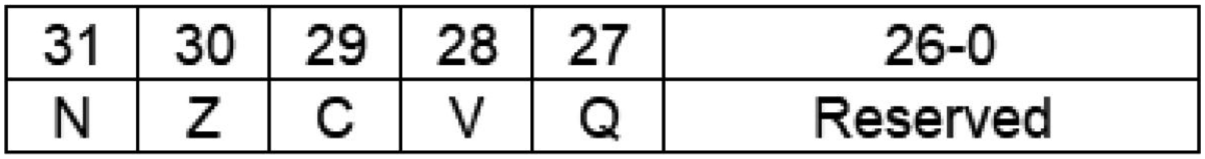
```
B label
```

which 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 –2,048 to 2,046. 2^{11} is 2,048, 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.

About the CPSR

The Current Program Status Register (**CPSR**) has been mentioned several times without really looking at what it contains. The carry flag was discussed when looking at the **ADD**/**ADC** instructions. In this section, a few more of the flags in the **CPSR** will be looked at.

All the flags it contains are shown in Figure 5-1, though a couple of them won't be discussed until later chapters. In this chapter, the group of condition code bits commonly used for conditional logic are studied.



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: Is set if the result is 0; this usually denotes an equal result from a comparison. If the result is non-zero, 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 **NE**	Z set Z clear	Equal 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

{condition}	Flags	Meaning
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, the **CMP** instruction needs to be looked at.

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, code the following:

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

In this context, 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. Studying Table 5-1 in this context should make the mnemonics make more sense.

Note **Rn** must be a low register (**R0–R7**); **Rm** can be any register (**R0–R15**). Both registers cannot be high registers.

Loops

With branch and comparison instructions in hand, constructing some loops modeled on what is found in high-level programming languages is looked at.

FOR Loops

Consider the **For** loop from the Basic programming language:

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

This can be implemented as shown in Listing 5-1.

```
      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
```

Listing 5-1 Basic For loop

To do this by counting down,

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

can be implemented as shown in Listing 5-2.

```
      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
```

Listing 5-2 Reverse For loop

Here an instruction is saved, since with the **SUB** instruction, the **CMP** instruction isn't needed.

WHILE Loops

To code a basic **While** loop:

```
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 Language, one possible implementation is shown in Listing 5-3.

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

Listing 5-3 While loop

> **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, how to implement the following pseudo-code in Assembly Language is considered:

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

In Assembly Language, the <expression> needs to be evaluated and the result placed in a register that can be used for comparison. For now, the following simple <expression> is considered:

```
register comparison immediate-constant
```

In this way, the expression can be evaluated with a single **CMP** instruction, for example, to code the following pseudo-code:

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

is implemented in Listing 5-4.

```
CMP R5, #10
       BGE elseclause

       ... if statements ...

       B endif
elseclause:

       ... else statements ...
```

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

Listing 5-4 If/then/else statement

This is 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

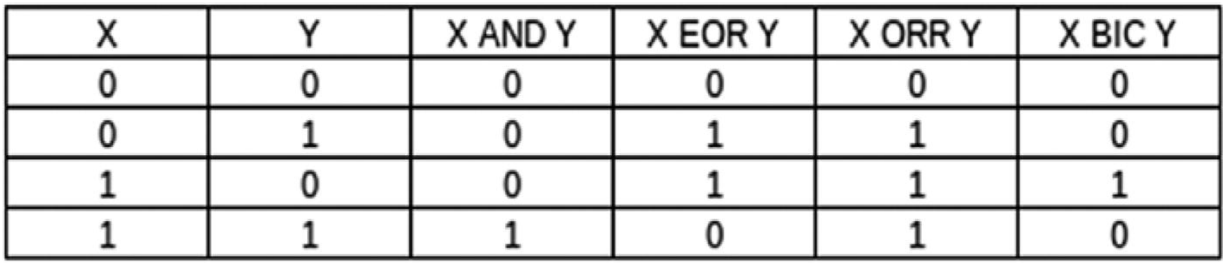
For the upcoming sample program, slightly more complexity is required, and it will start manipulating the bits in the registers. The ARM Cortex-M-series' logical operators provide several tools to do this, as follows:

- AND Rd, Rd, Rm
- EOR Rd, Rd, Rm
- ORR Rd, Rd, Rm
- BIC Rd, Rd, Rm
- MVN Rd, Rm
- TST Rn, Rm

These operate on each bit of the registers separately. Here are 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 each bit 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.

AND is often used to mask off a byte of information. Suppose only the high-order byte of a register is wanted. Listing 5-5 shows how to code this.

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

Listing 5-5 Using AND to mask a byte of information

This code will preserve the high-order byte while zeroing out the other 3 bytes. It takes 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 each bit 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 each bit 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 3 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 calculates 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 like the **CMP** instruction, but using **AND** instead of **SUB**, for example:

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

Design Patterns

When writing Assembly Language code, there is a great temptation to be creative. For instance, looping ten times could be done by setting the tenth bit in a register and then shifting it right until the register is zero. This works, but it makes reading the program difficult. If a program is put aside and returned to at a later date, the programmer will be scratching their head as to what the program does.

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

Design patterns make the programming more productive, since a collection of tried-and-true patterns for most situations is available to quickly utilize.

Tip In Assembly Language, make sure which design pattern being used is documented, along with documenting the registers used.

Therefore, loops and if/then/else are implemented in the pattern of a high-level language. If this is done, it makes the programs more reliable and quicker to write. Later, the macro facility in the GNU Assembler will be looked at to help with this.

Converting Integers to ASCII

The first example of a loop is to convert a 32-bit register to ASCII. In the HelloWorld program in Chapter 2, the Pico-series SDK's **printf** function was used to output the "Hello World" string. This program converts 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 comprise the alphabet, 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–Z are contiguous as are the numbers 0–9. 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 pseudo-code for what will be implemented in Assembly Language (Listing 5-6).

```
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 5-6 Pseudo-code to convert a register to ASCII

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

```
@ Example to convert contents of register to ASCII
@
@ 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"
```

Listing 5-7 Printing a register in ASCII

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 a debug build needs to be created with the UART set for printing. Remember the Debug Probe will translate this from UART to USB on the host computer. If not using VS code, then have the updated .**gdbinit** in place, and run **openocd** via the **ocdg** script.

Make sure the following code is understood and 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 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 the loop, **R4** is shifted 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 the original number is lost during this algorithm.

Most of the elements present in this program have already been discussed, 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. Including ASCII characters in immediate operands by putting them in single quotes.
2. Placing 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 the intent can be seen, rather than just having coded 55. There is no penalty to the program in doing this, since the work is done when the program is assembled, not when it is run.

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 memory indirection is being used and not just putting the byte into register **R1**. This is to make the program more readable, so this operation isn't confused with a corresponding **MOV** instruction.

Accessing data in memory is the topic of Chapter 6, where this is covered in far greater detail. The way the byte is stored could be made more efficient, and this will be looked at then.

Why Not Print in Decimal?

In this example program, the conversion to a hex string is simple 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. To convert to a decimal, base 10, string, then the program needs to be able to get the remainder from dividing by 10 and later divide by 10.

So far, division instructions haven’t been covered yet. This places converting to decimal beyond the scope of this chapter. A loop could be written to implement the long division algorithm learned in elementary school, but instead division is deferred until Chapter 12.

Performance of Branch Instructions

In Chapter 2, the ARM Cortex-M series instruction pipeline was discussed. Individually, an instruction requires three clock cycles to execute, one for each of the following instructions:

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 execution time is one instruction every clock cycle. But what happens when a branch occurs?

When the branch is executed, the next instruction is already decoded, and the instruction two ahead is loaded. When the branch happens, this work is thrown away, and the process starts over. This is seen 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 the branch isn’t taken and a **BL** instruction takes an extra cycle.

If a lot of branches are placed in the code, a performance penalty is suffered. Another problem is that when programming 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.

Summary

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

In Chapter 6, the details of how to load data to and from memory are described.

Exercises

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

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
```

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
```

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.

S. Smith, *RP2040 Assembly Language Programming*, Maker Innovations Series
https://doi.org/10.1007/979-8-8688-2202-5_6

6. Thanks for the Memories

Stephen Smith[1]
(1) Gibsons, BC, Canada

In this chapter, the memory of the Pico-series is explored in detail. Up to this point, memory has primarily served as a location to store Assembly Language instructions. The following sections provide an in-depth look at defining data in memory, loading data into registers for processing, and writing results back to memory.

The ARM Cortex-M series 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. The previous chapters mostly looked at the arithmetic and logical operations. Now it is time to look at the other category of load–store.

Memory addresses are 32 bits and instructions are 16 bits, presenting similar challenges to those discussed in Chapter 4, where various techniques were required to load 32 bits into a register.

In this chapter, these same techniques are applied to loading addresses, along with additional strategies. The objective is to load a 32-bit address in a single instruction whenever possible. However, before loading and building memory addresses, the contents of memory need to be defined with the GNU Assembler.

How to Define Memory Contents

The GNU Assembler contains several directives to help define memory to use in a program. These appear in a **.data** section of a program. First of all, some examples are looked at and then summarized in Table 6-1. Listing 6-1 shows how to define bytes, words, and ASCII strings.

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

Listing 6-1 Sample memory directives

The first line defines 7 bytes all with the same value. Numbers can be defined bytes in decimal, octal (base 8), binary, hex, or ASCII. Anywhere numbers are defined, expressions can be used that the Assembler will evaluate when it compiles a program.

Most memory directives start with a label, so they can be accessed symbolically from the code. The only exception is if defining a larger array of numbers that extends over several lines.

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

- A decimal integer starts with a non-zero 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 digits 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 a null-terminated ASCII string, as seen in the HelloWorld program in Chapter 1. There are two prefix operators that can be placed 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.

Here's an example:

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

Table 6-1 lists the various data types that can be defined this way.

Table 6-1 The list of memory definition Assembler directives

Directive	**Description**
.ascii	A string contained in double quotes
.asciz	A 0-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

To define a larger set of memory, there are a couple of mechanisms to use 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 following code

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

repeats the statements between **.rept** and **.endr**, 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
```

A rept/endr block can surround any Assembly Language code, for instance, to make a loop by repeating the code count times.

The special character "\n" for a new line was used in the HelloWorld string. 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 (e.g., \123)
\xdd	A hex ASCII code (e.g., \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. The Assembler can be instructed 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 data needs to be word aligned, then add the ".align 4" directive. This will result in 3 wasted bytes, but if this is a problem, this may need to be rearranged in the memory in the **.data** section.

ARM Cortex-M-series 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 the program will crash when it's run.

In the next section, when data is loaded with **PC**-relative addressing, those 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, the **LDR** instruction and its variations will be looked at. The **LDR** instruction is used 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 strategies to get as much as possible out of the 16-bit instructions. The cases will be examined 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**.

First, how to load or create a memory address in a register is looked at.

How to Load a Register with an Address

To create a memory address in a register, it can either be created from scratch or based on an address that is already in another register. First of all, the address is built directly.

How to Build the Address Directly

When a program is written under a modern operating system, like Linux, memory addresses can't just be created, because they need to be provided by the operating system to consider virtual memory and memory protection. On a microcontroller, like the RP2040 or RP2350, there is no operating system, virtual memory, memory management, or memory protection.

The memory map of the Pico-series is fixed and documented in the Pico-series SDK reference documentation. Therefore, there are many situations where the address is known ahead of time and needs to be loaded into a register. The previous chapter covered how to load a 32-bit register with any value, and this will work in this situation. Fortunately, many of the addresses that need to be dealt with are simple, such as 0xd0000014, which is the memory address to write for setting GPIO pins. Since most of the address is 0s, it can be loaded into a register with

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

Here, it 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 it is documented. Next, a more straightforward way of building addresses is looked at using an existing memory address in the program counter (**PC**).

PC-Relative Addressing

In Chapter 2, the **LDR** instruction was introduced to load the address of the "Hello World" string. This was needed to pass the address of what to print to the Pico-series 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 the data is kept close to the code, it is painless. The disassembly of the **LDR** instruction is shown below:

```
LDR R0, =helloworld
```

was formerly as follows:

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

Here is the instruction to load the address of the "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's a bit more complexity to this that will be addressed soon that makes this much more flexible.

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, to produce a range of 0–1,020.

Note This can also be done relative to the stack pointer (**SP**); however, the **SP** will be examined in detail in Chapter 7.

How to Load Data from Memory

In the HelloWorld program, the address was only needed to pass on to **printf**. Generally, these addresses are used 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 of all, **R1** is loaded with the address of the data wanted, and then that register is used to indirectly load register **R2** with the actual data.

```
      @ 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
```

Listing 6-2 Loading an address and then the value

Stepping through this in the debugger allows the process of loading 0x1234ABCD into **R2** to be watched step by step.

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

This works, but it took two instructions to load **R2** with the 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. This can be improved in some cases 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 can be loaded with one **LDR** instruction 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, this memory location cannot be written to when the program 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 the following:

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

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

```
// 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
```

Listing 6-3 Pseudo-code to loop through an array

The ARM Cortex-M-series instruction set provides support for doing these sorts of operations:

1. Define an array of ten words (4 bytes each):

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

2. Load the array's address into **R1**:

   ```
   LDR R1, =arr1
   ```

Elements of this array can be accessed using **LDR** as demonstrated in Listing 6-4 and graphically represented in Figure 6-1.

```
@ 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)]
```

Listing 6-4 Indexing into an array

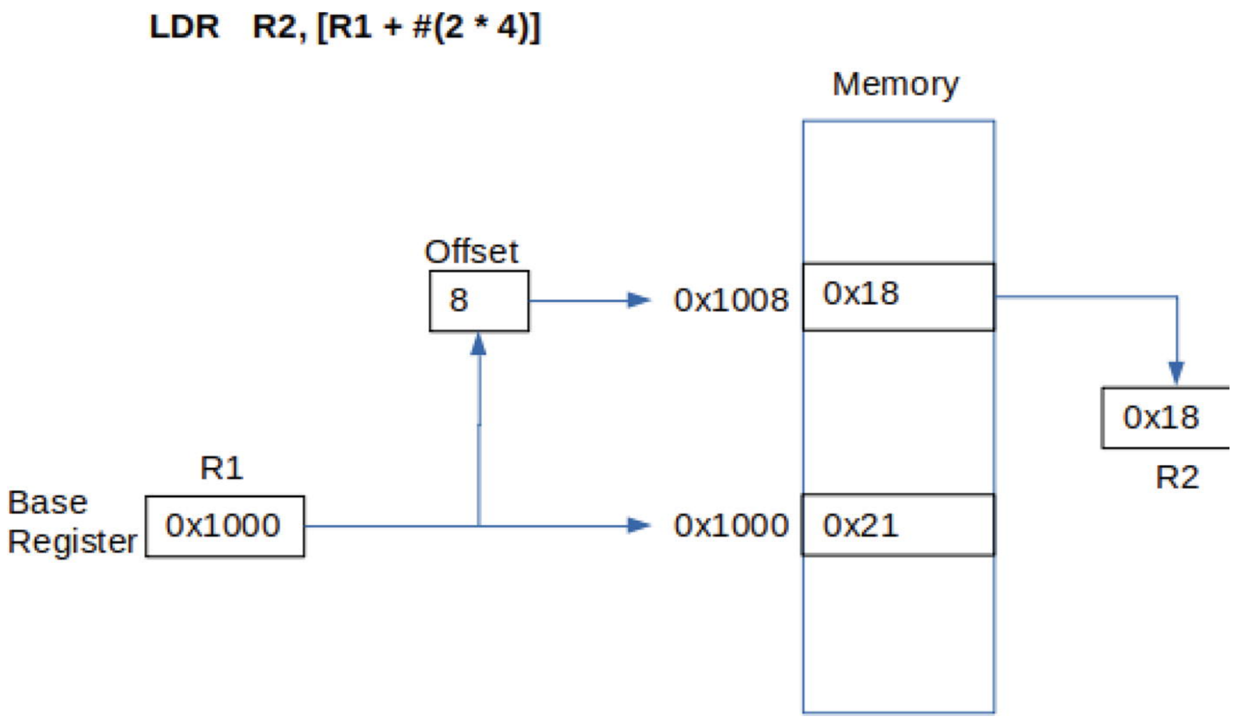


Figure 6-1 Graphical view of using **R1** and an index to load **R2**

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

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

Listing 6-5 Using a register as an offset

When incrementing through memory in a loop, increment either 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 the base memory address isn't destroyed by incrementing it.

Note The immediate value with the **LDR** instruction is only 8 bits, so can only be offset by 255 bytes. Consequently, 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 pseudo-code to do this.

```
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
```

Listing 6-6 Pseudo-code to convert a string to uppercase

This example uses NULL-terminated strings that are abundant in C programming. These were used for **printf** strings and were created with the .asciz directive. The string is the sequence of characters, followed by a NULL (ASCII code 0 or \0) character. To process the string, simply loop until the NULL character is encountered.

For and **While** loops have already been covered. 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. This is desired, since if the string is empty the NULL character still needs to be copied, 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 pseudo-code.

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

Listing 6-7 Pseudo-code on how we will implement the IF statement

Listing 6-8 is the Assembly Language code 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
```

Listing 6-8 Program to convert a string to uppercase

Note The provided source code combines samples of using data directives, along with the various forms of the LDR and STR instructions along with the uppercase conversion program.

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 the pointers for **instr** and **outstr.**
- **Five instructions**: Make up the **if** statement.
- **Six instructions**: For the loop, including loading a 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, there's the need for the **CMP** instruction in the **UNTIL** part of the loop to test for NULL. In this example, the **LDRB** and **STRB** instructions are used since the string is processed byte by byte. To convert the letter to uppercase, 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 the program.

```
100002b6:      4c08         ldr   r4, [pc, #32]      @
(100002d8 <cont+0x10>)
100002b8:      4b08         ldr   r3, [pc, #32]      @
(100002dc <cont+0x14>)

100002ba <loop>:
100002ba:      7825         ldrb  r5, [r4, #0]
100002bc:      3401         adds  r4, #1
100002be:      2d7a         cmp   r5, #122            @
0x7a
100002c0:      dc02         bgt.n     100002c8 <cont>
100002c2:      2d61         cmp   r5, #97             @
0x61
```

```
100002c4:      db00         blt.n     100002c8 <cont>
100002c6:      3d20         subs  r5, #32

100002c8 <cont>:
100002c8:      701d         strb  r5, [r3, #0]
100002ca:      3301         adds  r3, #1
100002cc:      2d00         cmp   r5, #0
100002ce:      d1f4         bne.n     100002ba <loop>
100002d0:      4802         ldr   r0, [pc, #8]        @
(100002dc <cont+0x14>)
100002d2:      f002 fec1    bl        10003058
<__wrap_printf>
100002d8:      200005e7     .word     0x200005e7
100002dc:      20000616     .word     0x200006162000025f
<instr>:
2000028e <outstr>:
```

Listing 6-9 Disassembly of the uppercase program

The instruction is as follows:

```
LDR R4, =instr
```

is converted to the following:

```
ldr   r4, [pc, #32]      @ (100002d8 <cont+0x10>)
```

The comment documents that **PC**+32 is the address 0x100002d8. This is calculated by taking the address of the next instruction (the one being decoded as this one executes), which is at 0x100002b8, and adding 32 to get the same 0x100002d8.

This shows how the Assembler added the literal for the address of the string **instr** at the end of the code section. When the **LDR** is executed, it accesses this literal and loads it into memory, and this provides the address needed. 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**. Watch the registers with the "i r" (info registers) command. To view **instr** and **outstr** as the processing occurs, there are a couple of ways of doing it. From the disassembly, the address of **instr** is 0x200005e7, so it can be viewed with

```
(gdb) x /2s 0x200005e7
```

```
0x200005e7: "This is our Test String that we will
convert.\n"
0x20000616: "THI"
(gdb)
```

This is convenient since the **x** command knows how to format strings, but it doesn't know about labels. An alternative code is as below:

```
(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. The label must be cast 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. Next, two instructions for loading and storing multiple registers at once are examined.

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 is 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, **R2** and **R3** are stored back into **mydword2**.

```
      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
```

Listing 6-10 Example of loading and storing multiple registers

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, an extra **ADD** instruction isn't needed 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, data can be loaded from memory, operated on in the registers, and then saved back to memory. How the data load and store instructions help with arrays of data and how they help us index through data in loops were examined.

In the next chapter, how to make code reusable is looked at. After all, wouldn't the uppercase program be handy if it could be called whenever needed?

Exercises

1. Create a small program to try out all the data definition directives the Assembler provides. Assemble the program and examine the data in the disassembly listing. Add some align directives and examine how the data moves around.
2. Explain how the **LDR** instruction lets any 32-bit address load in only one 16-bit instruction.
3. Write a program that converts a string to all lowercase.
4. Write a program that converts any non-alphabetic character in a NULL-terminated string to a space.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
S. Smith, *RP2040 Assembly Language Programming*, Maker Innovations Series
https://doi.org/10.1007/979-8-8688-2202-5_7

7. Calling Functions and Using the Stack

Stephen Smith[1]
(1) Gibsons, BC, Canada

This chapter explores methods for organizing code into independent units known as **functions**. In software development, the process often begins with low-level components, which serve as the foundation for building more complex applications. Reusable components can be called from any part of a program, promoting modularity and clarity. Previous lessons covered looping, conditional logic, and arithmetic operations. Now, attention turns to compartmentalizing code into effective building blocks called stacks.

Stacks, a fundamental data structure in computer science, are used for storing data on an as-needed basis. Building useful and reusable functions requires a method for managing register usage so that functions do not interfere with each other. Chapter 6 discussed storing data in main memory, which persists for the duration of the program. Small functions, such as those for converting strings to uppercase, may need a few memory locations during execution, but this memory is no longer required once the function completes. Stacks offer a solution for managing register usage across function calls and supplying memory to functions only for the length of their invocation.

Several low-level concepts are introduced at the outset, followed by the process of combining them to effectively create and use functions. The explanation begins with stacks and their implementation on the Pico-series.

About Stacks on the Pico-series

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 on the M-series CPU, the size of the stack is configurable, by default 0x800 (2,048 words). In Chapter 2, it was mentioned that register **R13** had a special purpose as the stack pointer (**SP**). This is why **R13** is named **SP** in **gdb**, and typically it has 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. Any low register (**R0–R7**) as well as **LR** can be included in the **PUSH** instruction and **PC** in the **POP** instruction. Why this functionality for **LR** and **PC** is useful will be seen 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.

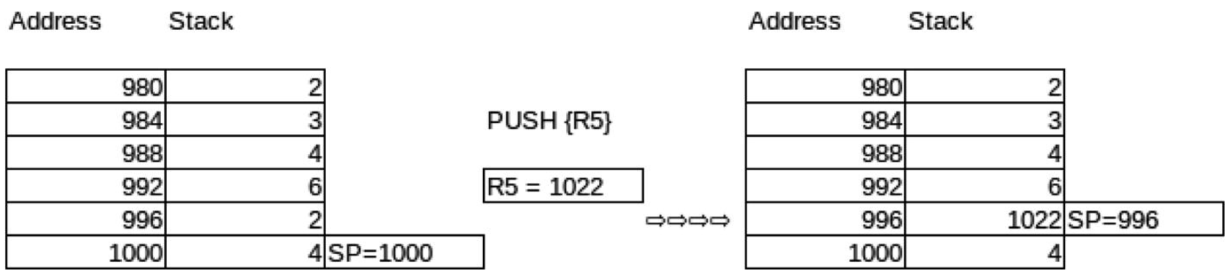


Figure 7-1 Pushing R5 onto the stack

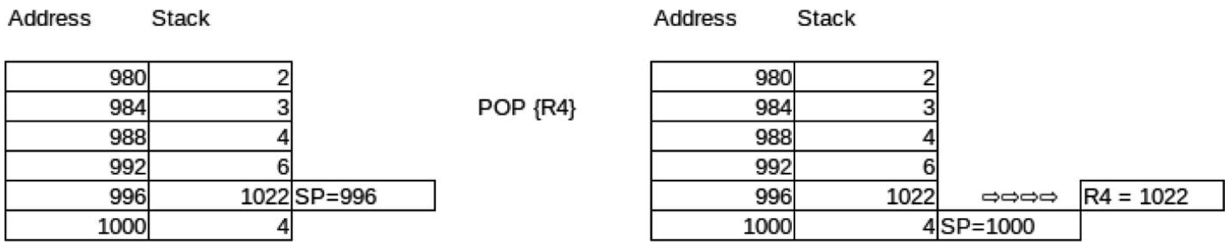


Figure 7-2 Popping R4 from the stack

Before making use of these instructions, calling and returning from functions need to be looked at.

How to Branch with Link

To call a function, the ability for the function to return execution to the instruction after the point where the function was called is needed. 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.

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

Listing 7-1 Skeleton code to call a function and return

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

About Nesting Function Calls

A function was successfully called and returned from, but the stack was never used. Why introduce the stack first and then not use it? First of all, think of what happens if during its processing **myfunc** calls another function. This is fairly common, as code is written building on the functionality previously developed.

If **myfunc** executes a **BL** instruction, then **BL** copies the next address into **LR** overwriting the return address for **myfunc**; however, **myfunc** won't be able to return. What's needed is a way to keep a chain of return addresses as function after function is called. 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 **LR** can be **PUSH**'ed but not **POP**'ed. Instead, the **PC** can be **POP**'ed. The reason is that this saves 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, demonstrating how convenient it is to store data to the stack that only needs to exist for the duration of a function call.

```
     @ ... 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
```

Listing 7-2 Skeleton code for a function that calls another function

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 versus maintainability is a subjective decision depending on the circumstances.

When working in high-level programming languages, 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 accept parameters and return results, and Assembly Language programming operates similarly. Inventing custom mechanisms for parameter passing and result returning can prove counterproductive. Code often needs to interoperate with other programming languages. For example, it may be necessary to call new, efficient functions from C code or to invoke functions written in C, such as those provided by the Pico-series SDK.

To facilitate this, there is a set of design patterns for calling functions. All the code that follows these patterns will be able to interoperate freely.

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 there are only two parameters, then **R0** and **R1** would be used. 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 more data needs to be returned, then have one of the parameters be an address to a memory location where the additional data can be placed. This is the same as C when it returns data through call by reference parameters.

The ARM M-series CPU only contains 16 registers, and most instructions only work with eight of these. How then to ensure that the calling function's registers aren't wiped out when a function is called? This is the topic of the next section.

How to Manage the Registers

If a function is called, chances are it was written by a different programmer, and what registers it uses may not be known. It would be very inefficient if every register needed to be saved and restored every time a function is called. As a result, there is 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 it is responsible for saving them. That means the calling routine can assume these registers are intact.
- **R12**: This is the intra-procedure call scratch register and shouldn't be used. Some SDK functionality (like **printf**) will not work if this register is modified.
- **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 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, the function call algorithm can be summarized.

Summary of the Function Call Algorithm

Calling routine:

1. If any of **R0–R4** are needed, 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. Perform the function body.
3. Put the 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 some of these registers aren't used, 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.

All parameters are assumed to be 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 all the branch instructions have been introduced, some extra, perhaps unexpected, functionality needs to be noted.

More on the Branch Instructions

These are the branch instructions supported by the ARM Cortex-M-series 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 –2,048 to 2,046 from the current program location. This makes them appropriate for loops and jumps within single functions. This prevents writing large single routines, which jump madly about.
- Number 4 is one of the 32-bit instructions supported by the ARM Cortex-M-series. This is a **PC**-relative offset, but the range is –16,777,216 to 16,777,214, which is larger than the amount of memory contained in either SRAM or flash on all current Pico-series boards. This means any routine in the program or the SDK can be called 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 Pico-series' full 32-bit address space.

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

About the X Factor

The **BX** instruction is called the Branch and Exchange instruction, which begs the question: what is being exchanged? In the full ARM A-series processors, like those used in the Raspberry Pi 5, when running in 32-bit mode, there are two separate sets of instructions:

1. The regular 32-bit-long instructions
2. The 16-bit “thumb” instructions that 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 ARM M-series CPU only supports type 2 instructions, but there is only one instruction set, so why discuss this? The problem to be careful of is that if **BX** or **BLX** thinks that instruction set type 1 is being switched to, then the Pico-series CPU throws a hardware fault, and the program terminates.

Since all instructions must 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 instructions 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 nothing is done then the Assembler generates even addresses and the M-series CPU generates a hardware fault when it tries to jump. This is why

```
.thumb_func
```

must be placed before the definition of every function called by **BX** or **BLX**.

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.

In the uppercase function studied next, 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

Next, the uppercase example from Chapter 6 is reorganized as a proper function. The function is moved into its own file, and **CMakeLists.txt** is modified to include both the calling program and the uppercase function. First, create a file called **main.S** containing Listing 7-3 for the driving application.

```
@
@ 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
```

Listing 7-3 Main program for the uppercase example

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

```
@
@ Assembly Language 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
```

Listing 7-4 Function to convert strings to all uppercase

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

```
cmake_minimum_required(VERSION 3.13)
set(PICO_BOARD pico2 CACHE STRING "Board type")

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)
```

Listing 7-5 CMakeLists.txt for the uppercase function example

Step through the function call to examine the contents of important registers and the stack. Set a breakpoint at main and single step through the first couple of instructions and stop at the **BL** instruction. The program sets **R4** to 12 and **R5** to 13, to make it easy to follow how these are saved to the stack.

```
R4     0xc                  12
R5     0xd                  13
sp     0x20082000           0x20082000
lr     0x10003093           268447891
pc     0x10000280           0x10000280 <repeat+8>
```

Notice the **BL** instruction is at 0x10000280. Now single step again to execute the **BL** instruction. Here are the same registers:

```
R4     0xc                  12
R5     0xd                  13
sp     0x20082000           0x20082000
lr     0x10000285           268436101
pc     0x100002ea           0x100002ea <toupper>
```

The **LR** has been set to 0x10000285, 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 to continue to use 16-bit instructions. The **PC** is now 0x100002ea, pointing to the first instruction in the toupper routine. The first instruction in toupper is the **PUSH** instruction to save registers **R4** and **R5**. Single step through that instruction and examine the registers again.

```
R4     0xc                  12
R5     0xd                  13
sp     0x20081ff8           0x20081ff8
lr     0x10000285           268436101
pc     0x100002ec           0x100002ec <toupper+2>
```

The stack pointer (**SP**) has been decreased by 8 bytes (two words) to 0x20081ff8. None of the other registers have changed. **PUSH**'ing registers onto the stack does not affect their values; it only saves them. Looking at location 0x20081ff8 reveals the following:

```
((gdb) x /4xw 0x20081ff8
0x20081ff8: 0x0000000c 0x0000000d 0x00000000
0x00000000
```

Copies of registers **R4** and **R5** are now on the stack, and **SP** points to the last item saved (and not the next free slot).

Note The toupper function doesn't call any other functions, so **LR** is not saved along with **R4** and **R5**. If another function is ever called, then **LR** will need to be added to the list. This version of toupper is intended to be as fast as possible, so no extra code is added 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 to 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", which 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 the uppercase function, additional memory wasn't needed since all the work could be done with the available registers. When larger functions are coded, more memory is often required for the variables. Rather than add clutter to the **.data** section, these variables can be stored on the stack.

PUSH'ing these variables on the stack isn't practical, since they usually need to be accessed 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 three variables are needed, each 32-bit integers, say, a, b, and c. Therefore, 12 bytes need to be allocated on the stack (3 variables × 4 bytes/word).

```
SUB SP, #12
```

This moves the stack pointer down by 12 bytes, providing a region of memory on the stack to place the variables. Suppose a is in **R0**, b in **R1**, and c in **R2**, these can then be stored 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, the following needs to be executed

```
ADD SP, #12
```

to release the variables from the stack. Remember, it is the responsibility of a function to restore **SP** to its original state before returning. Next, an example is presented.

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.

```
@ 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
```

Listing 7-6 Simple skeletal function that demonstrates a stack frame

A new concept is introduced in this example—symbols via the **.EQU** directive.

How to Define Symbols

This example introduced the **.EQU** Assembler directive. This directive allows the definition of symbols that will be substituted by the Assembler before generating the compiled code. This way, the code can be made more readable. Otherwise, keeping track of which variable is which on the stack makes the code hard to read and is error-prone. With the **.EQU** directive, each variable's offset is defined once. Sadly,

.EQU only defines numbers, so it can't be used to define the whole "[SP, #4]" string.

Functions aren't the only way to make reusable code. Next, macros are looked at.

How to Create Macros

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

```
@
@ 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}
.data
tststr:  .asciz  "This is our Test String that we
will convert.\n"
tststr2: .asciz  "A second string to upper
case!!\n"
buffer:  .fill      255, 1, 0
```

Listing 7-7 Program to call the toupper macro

The **mainmacro.S** code is set up as a function and called from **main.S** with

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

This way only one project is needed 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.

```
@
@ 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
pointer
@ 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
```

Listing 7-8 Macro version of the toupper function

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

About the Include Directive

The file **uppermacro.S** defines the 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 the source file becomes larger. This is done before any other processing. This is like the C **#include** preprocessor directive.

Now that a mechanism to include macros is set, how to define macros is looked at next.

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, parameters are specified 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
```

How the parameters are used in the code can be seen with \instr and \outstr. These are text substitutions and need to result in correct Assembly Language syntax, or an error will result. 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 the original labels were left in place, an error that a label is defined more than once would occur if the macro is used more than once. The strategy here is that the Assembler lets numeric labels be defined as many times as needed. To reference them in the code, use

```
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, toupper is called twice in **mainmacro.S** to show that everything works and that this macro can be reused as many times as needed. But why 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 notice 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 because of performance. The Pico-series runs at 133MHz or 150MHz, which isn't that fast by modern standards. Remember that whenever a branch is taken, the execution pipeline needs to be restarted, making branching an expensive instruction. With macros, the **BL** branch to call the function is eliminated along with the **BX** branch to return. The **PUSH** and **POP** instructions to save and restore any registers used are also eliminated. 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

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

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

Next, in Chapter 8, more detail are provided about calling and being called by C routines, in particular, how to interact with the Pico-series SDK.

Exercises

1. Suppose a function 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 the high registers **R8** and **R9** are handled.
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.
3. Convert the lowercase program in Exercise 2 to a macro. Have it run on the same three test strings to ensure it works properly.
4. Why does the function calling protocol have some registers that need to be saved by the caller and some by the callee? Why not make all saves by one or the other?
5. Why would the SDK call the main routine with a **BLX** instruction rather than a **BL** instruction?

S. Smith, *RP2040 Assembly Language Programming*, Maker Innovations Series
https://doi.org/10.1007/979-8-8688-2202-5_8

8. Interacting with C and the SDK

Stephen Smith[1]
(1) Gibsons, BC, Canada

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 possible out of the hardware. Modern compilers, like the GNU C compiler, generate adequate code, and microcontrollers, like the Pico-series, are much faster. As a result, most applications are written in a collection of programming languages, where each excels at a specific function.

The Pico-series SDK contains a wealth of efficient code, offering extensive resources that can be leveraged instead of developing everything from scratch. Most of the SDK is implemented in C, with numerous Assembly Language routines available for further study.

This chapter looks at using components written in C/C++ from Assembly Language code and at how other languages can make use of the fast-efficient code being written in Assembly Language.

This chapter uses the Raspberry Pi Pico-series' hardware I/O capabilities. How to set up three flashing LEDs is described, and then how to control them using different techniques is covered over the following two chapters. This chapter shows how to control the LEDs using the Pico-series SDK. This provides more experience using C functions and the extra complexity present in the SDK.

How to Wire Flashing LEDs

Before writing programs, circuitry needs to be wired to connect three LEDs to a breadboard. This project requires

- Three 220Ω resistors (red, red, black)
- Three LEDs (preferably of different colors, such as red, blue, and green)
- Four connecting differently colored wires

These instructions assume that the pins are already soldered to the Pico-series board that has been plugged into a breadboard as outlined in Chapter 1. These parts are typically included in any Raspberry Pi or Arduino electronics starter kit.

Each of three LEDs is connected to a GPIO pin, in this case 18, 19, and 20, and then to ground through a resistor. The resistor is needed because the GPIO is specified to keep the current under 16mA, or the circuits can be damaged. Most kits come with several 220Ω 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 13Ω and variable).

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

Note The GPIO pins are numbered differently internally and externally. When the program accesses GPIO 18 internally, this is wired to the external pin 24 on a Raspberry Pi Pico 2 board. Check the pinout diagram for whichever board being used, to ensure the correct pin is wired up.

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

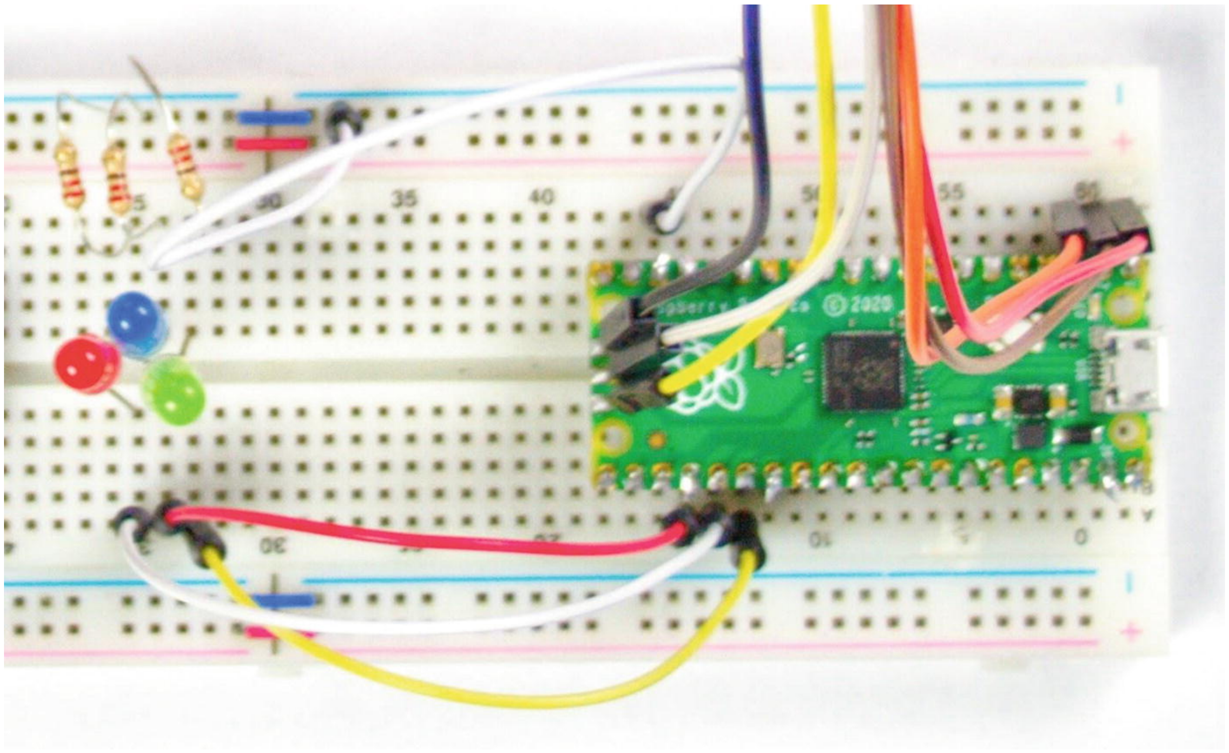


Figure 8-1 Breadboard with LEDs and resistors installed

Figure 8-2 shows a schematic of the flashing LEDs hardware to help with setting it up.

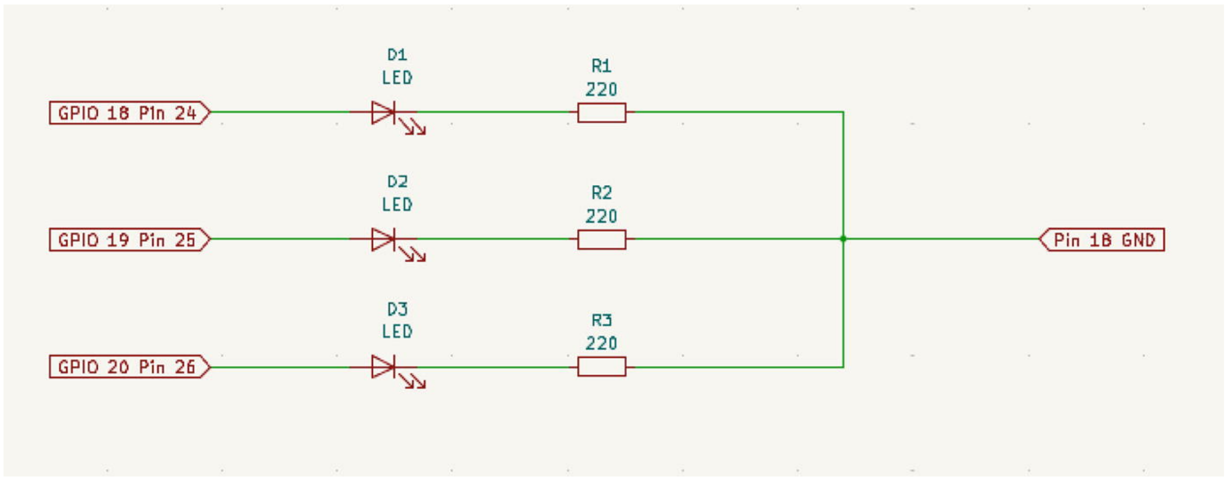


Figure 8-2 Schematic for the flashing LEDs

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

How to Flash LEDs with the SDK

In this chapter, the LEDs are flashed using functions in the Pico-series SDK. In later chapters, this process is repeated using Assembly Language to write to the hardware directly and then using the Pico-series' PIO coprocessors to offload the work from the CPU. Using the SDK is easiest, since it provides well-tested, ready-to-use functions that hide the complexities of directly interacting with hardware devices. Later parts of the program that aren't performant can be identified and rewritten in Assembly Language or use coprocessors to create a better experience.

In this example, four SDK functions are used:

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 covered in Chapter 7; therefore, the first parameter is placed in **R0** and the second parameter in **R1**. None of these functions return a value, so **R0** doesn't need to be checked after making the call. Basically, the following is done:

1. Initialize the three GPIO pins 18, 19, and 20.
2. Sequentially turn on a LED.
3. Sleep for one-fifth 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**.

```
@
@ 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
```

Listing 8-1 Assembly Language source code to flash the LEDs using the SDK

This program calls **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 similar to the 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.

```
/* 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);
}
```

Listing 8-2 C wrapper functions for the inline code we need from the SDK

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.

```
cmake_minimum_required(VERSION 3.13)

set(PICO_BOARD pico2 CACHE STRING "Board type")
```

```
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)
```

Listing 8-3 CMakeLists.txt file for this project

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

New approaches to functions like **gpio_put** will be covered in the following chapters, but initialization functions like **gpio_init** are typically not time critical and using the SDK function is fine.

How to Call Assembly Routines from C

A typical scenario is to write most of an application in C and then call Assembly Language routines in specific time-critical use cases. Following the function calling protocol from Chapter 7, C won't be able to tell the difference between Assembly Language functions and any other functions written in C.

An example is to call the **toupper** function from Chapter 7 from C. Listing 8-4 contains the C code for **uppertst.c** to call this Assembly Language function.

```
//
// C program to call the 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);
      }
}
```

Listing 8-4 Main program to show calling the toupper function from C

The name of the toupper function is changed to **mytoupper,** since there is already a toupper function in the C runtime. Without this change a multiple-definition error results. 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 the function to the C compiler with

```
extern int mytoupper( char *, char * );
```

This should be familiar to all C programmers; this must be done for C functions as well. Usually, all these definitions are gathered together and put into a header (.h) file.

When the program is run, the string is converted to 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 this routine is used a lot with C, subtract 1, so that the 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 the core algorithm for the toupper function is embedded inside the C program.

```
//
// C program to embed the 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);
    }
}
```

Listing 8-5 Embedding our Assembly routine directly in C code

The **asm** statement allows Assembly Language code to be embedded directly into C code. With this, an arbitrary mixture of C and Assembly Language code can be written. 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 the routine, in this case "r" (str); "r" (outBuf) means two registers are needed—one holds **str** and one holds **outBuf**. It is fortunate that C string variables hold the address of the string, which is what is needed 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 a C label is jumped to, warn the compiler with a **goto asm-qualifier**.

The input and output operands can be labeled; this wasn't done, which means the compiler will assign names **%0, %1**, … as used in the Assembly Language code.

If the program is disassembled, notice 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 **outstr** can't be modified. For **outstr,** since its value was saved to **R4** for the length calculation, it can be restored at the end. **instr** is moved into **R0** and incremented there, so that the input register is preserved.

Note If too many registers are specified, then the inputs will be received 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 this code and the optimizer might remove important code.

Summary

This chapter studied calling C functions from Assembly Language code. The functions in the RP2040's SDK were used to access the GPIO pins, and how to deal with inline C functions was covered. Then the reverse of calling the Assembly Language uppercase function from a C main program was written. Next, the Assembly Language code was embedded directly inline into C code.

Accessing the RP2040's hardware indirectly through the SDK works and is quick, but Assembly Language programmers like to access the hardware directly, which is the topic of Chapter 9.

Exercises

1. Create a C program to call the lowercase routine from Chapter 7, Exercise 2, and print out some test cases.
2. Take the lowercase routine from Chapter 7, Exercise 2, and embed it in C code using an **asm** statement.
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**.
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?

S. Smith, *RP2040 Assembly Language Programming*, Maker Innovations Series
https://doi.org/10.1007/979-8-8688-2202-5_9

9. How to Program the Built-In Hardware

Stephen Smith[1]
(1) Gibsons, BC, Canada

Chapter 8 interacted with external hardware devices connected to the GPIO pins using the Pico-series SDK. This chapter looks at interacting with the hardware directly. No new Assembly Language instructions need to be learned, because access to the hardware is accomplished with the memory **load/store** instructions previously studied. All hardware access is via special memory addresses connected to hardware devices, which 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 memory addresses directly, a lay of the land is needed. This chapter gives details about the Pico-series' memory map.

About the Pico-series Memory Map

The RP2040/RP2350 contains several types of memory plus a large selection of hardware memory-mapped registers:

- Two banks of read-only memory
- 264kb or 520kb of read–write memory
- Several large banks of memory-mapped hardware registers that control the hardware or send/receive data to/from it

Table 9-1 is a high-level map of the main memory areas.

Table 9-1 High-level memory map of the RP2040/RP2350

Base Address	**Purpose**
0x00000000	On-chip boot ROM
0x10000000	Off-chip flash memory 16MB max
0x20000000	On-chip SRAM
0x40000000	Hardware registers for peripherals connected to the Advanced Peripheral Bus (APB) Bridge
0x50000000	Hardware registers for devices connected to AHB
0xd0000000	Hardware registers connected directly to CPU such as SIO
0xe0000000	Arm Cortex-M-series processor hardware registers

When looking 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 the program between power resets and is what the boot loader will run on power-up. The data variables and the stack are in the 0x20000000 range, indicating these aren't stored between power resets, but are easy to write to. The memory addresses from the other sets will be used shortly. This is how the programs view the various hardware devices connected to the Pico-series as special memory addresses. Next, a friendlier way to refer to these memory addresses and registers is looked at.

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. These don't need to be defined in the code 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 the code is written in Assembly Language. How can C header files be used?

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 only 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 Pico-series 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. How this helps will be seen shortly.

These are the basics for programming access. Next, hardware devices are connected to the outside world via the pins exposed on the boards, specifically, to the Raspberry Pi Pico-series. For directions on how to connect other manufacturers' RP2040/RP2350 boards, refer to their documentation.

About the Raspberry Pi Pico Pins

Notice that in a pinout for the Raspberry Pi Pico's external pins, each pin is labeled with several functions. The various peripherals contained in the RP2040/RP2350 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 Descriptions 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, multi-master, multi-slave, 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. It is 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 several internal clock domains on the RP2040/RP2350.
CLOCK GPOUT	General-purpose clock outputs. Can drive several 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.

2. Add that to the base to get the registers for the desired pin. This gives 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 IO_BANK0_GPIO3_ CTRL_FUNCSEL_VALUE_SIO_3 (value 5) from io_bank0.h to the control register.

 The code to do this follows in Listing 9-1.

```
#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
```

Listing 9-1 Code to set the GPIO pin to the SIO function, where the pin is provided in R0

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 Pico-series provides help to make programming hardware registers easier, which is shown next.

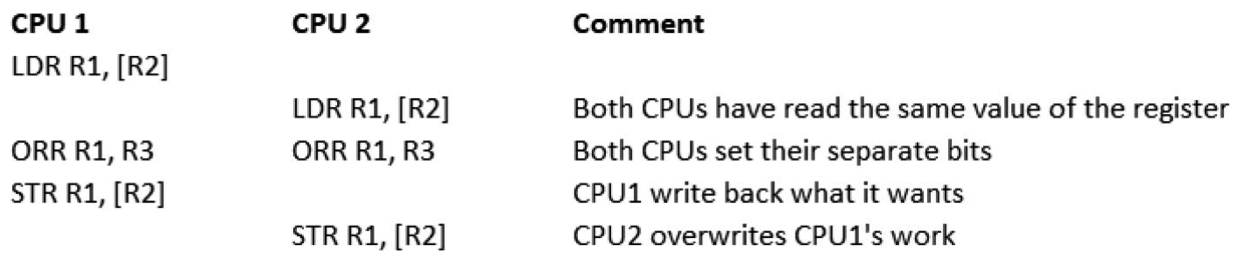
About Hardware Registers and Concurrency

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 resembles

```
LDR   R1, [R2]      @ R2 is the address of the
hardware register
ORR   R1, R3        @ R3 has one bit set
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 Pico-series 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 write back what it wants
	STR R1, [R2]	CPU2 overwrites CPU1's work

Figure 9-1 Flow of two CPUs with a concurrency problem

The Pico-series 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, use the set register. Each bit is for a different pin. Just write a value to the set register, where any one bit in the 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 zeroes are ignored.

This scheme is the reason for the name SIO for single-cycle I/O, since only one instruction is needed and thus one clock cycle sets or clears an I/O pin. On some pins there is also an **XOR** register, which 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, which add SIO functionality. 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.

```
      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
```

Listing 9-2 How to configure a pad

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.

About RP2350 Pad Isolation

The RP2350 introduced pad isolation, which isolates the pad electrically when the CPU is changing power states. When initialized the pads are electrically isolated from the outside world. When the pads are configured, they need to have this isolation removed before they can be used. To do this the isolation bit in the pad control register needs to be cleared. This is done with the code in Listing 9-3.

```
#if HAS_PADS_BANK0_ISOLATION
@ Remove pad isolation now that the correct
peripheral is set
       LDR   R2, padsbank0
       LSL   R3, R0, #2      @ pin * 4 for
register address
       ADD   R2, R3          @ Actual set of
registers for pin
       LDR   R4, clearoffset
       ADD   R2, R4
       LDR   R1, PBGIB
       STR   R1, [R2, #PADS_BANK0_GPIO0_OFFSET]
#endif
```

Listing 9-3 Code to remove pad isolation

Notice the #if statement, which the SDK CMake system will define HAS_PADS_BANK0_ISOLATION for any board with pad isolation such as the RP2350. Using this if allows the code to be complied and to work for either the Pico 1 or Pico 2.

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 0–28 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 9-4 shows this process.

```
#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
```

Listing 9-4 How to configure the SIO pin to a known state

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 9-5.

```
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]
```

Listing 9-5 Code to turn on a LED by turning on the SIO output register

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 9-6. 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.

```
@
@ 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 FUNCSEL_VALUE_SIO, 5

        .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 and 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.
         MOV   R4, R0
         LSL   R4, #3                  @ each GPIO
has 8 bytes of registers
         LDR   R2, iobank0             @ address
we want
         ADD   R2, R4                  @ add the
offset for the pin number
         MOV   R1, #FUNCSEL_VALUE_SIO
         STR   R1, [R2,
#IO_BANK0_GPIO0_CTRL_OFFSET]
#if HAS_PADS_BANK0_ISOLATION
@ Remove pad isolation now that the correct
peripheral is set
         LDR   R2, padsbank0
         LSL   R3, R0, #2              @ pin * 4
for register address
         ADD   R2, R3                  @ Actual
set of registers for pin
         LDR   R4, clearoffset
         ADD   R2, R4
         LDR   R1, PBGIB
         STR   R1, [R2, #PADS_BANK0_GPIO0_OFFSET]
#endif
         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
clearoffset:.word   REG_ALIAS_CLR_BITS
padenaboff: .word   PADS_BANK0_GPIO0_OFFSET
PBGIB:      .word   PADS_BANK0_GPIO0_ISO_BITS
```

Listing 9-6 The complete program to flash the LEDs writing to the hardware directly

The SDK **gpio_init** function defaults setting the SIO pin for input, so **gpio_set_dir** needed to be called to set the pin for output. In this example, the included **gpioinit** function sets the pin for output so the extra function isn't required.

Summary

This chapter studied how the memory in the Pico-series is organized, where ROM, RAM, and the hardware registers are located. How to use the C header files in the SDK to get symbolic references for the hardware registers and their values was learned. How the internal hardware devices are connected to external pads that we soldered pins to was studied. The APB was programmed to connect pins and make the SIO pins used active. The SIO registers were used to turn the LEDs on and off. The chapter concluded with an Assembly Language version of Chapter 8's program 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-M-series processor. In Chapter 10, how to offload this work to the Pico-series’ I/O coprocessors is studied.

Exercises

1. What is the starting memory address for the hardware registers for I2C number 0 I/O device? Which header file is looked in for useful defines when working with this device?
2. Why does the Raspberry Pi Pico-series have multiple functions on each external pin? Why doesn’t the Pico-series just have more pins so they can all be used at once?
3. Try changing the program to flash the LEDs in a different pattern. Can a fourth and fifth LED be added?
4. To make sure how the program loads the hardware addresses is understood, 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.
5. How would the program be structured to do other work, rather than calling **sleep_ms()**?

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
S. Smith, *RP2040 Assembly Language Programming*, Maker Innovations Series
https://doi.org/10.1007/979-8-8688-2202-5_10

10. How to Initialize and Interact with Programmable I/O

Stephen Smith[1]
(1) Gibsons, BC, Canada

This chapter puts aside the Assembly Language instructions for the ARM Cortex-M-series processor that have been studied and looks 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, whereas the RP2350 has twelve of these. There's a tool in the SDK, pioasm, which assembles these in a similar manner to the GNU Assembler.

The RP2040 and RP2350 contain several specialized I/O hardware components for handling various common hardware protocols like the UART and USB. However, with DIY projects non-standard devices are often encountered that require custom control of the GPIO pins. Sometimes it's possible to implement these protocols using the ARM CPU in a manner as in Chapter 9, but the ARM CPU wasn't designed for this, and it takes all the ARM's processing power if it's even possible.

Raspberry's solution to this are 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.

The good news is that there are only nine Assembly Language instructions, and there are only thirty-two 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 150MHz for the RP2350. 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.

Note The RP2350 is slightly faster than the RP2040 running at 150MHz rather than 120MHz. As a result, the sample programs flash the LEDs a bit faster on an RP2350 than on an RP2040, and the sample calculations in this chapter are based on the RP2040, but can easily be adapted to the RP2350.

Before diving into an example, the architecture of the PIO system is looked at first.

About the PIO Architecture

The PIO coprocessors are divided into banks of four—the RP2040 has two banks and the RP2350 has three banks. 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.

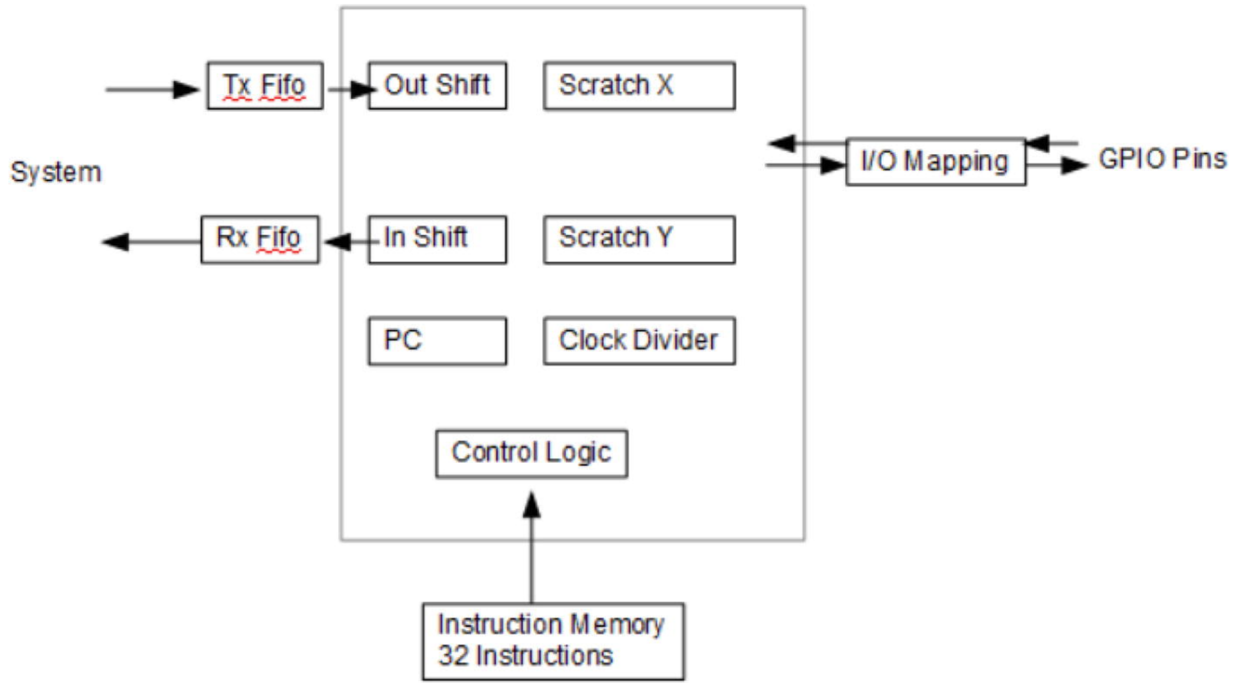


Figure 10-1 Block diagram of one PIO processor

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 (First-In-First-Out) 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’s 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 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 that 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, the nine individual instructions are looked at.

About the PIO Instructions

This section looks at the nine instructions and their operands. All these instructions can have a side-set or delay value included, but for simplicity these are looked at 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 going into detail on these instructions, an example follows to give a feel for how these instructions are used.

Flashing the LEDs with PIO

The previous programs flashed three LEDs with the SDK and then wrote directly to the Pico-series' hardware registers, but now the PIO coprocessor is used. The advantage of this method is that all the processing happens on three PIOs and the ARM processor is left free to do other work. First, the PIO Assembly Language code is put in a file called **blink.pio** containing Listing 10-1.

```
;
; 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);
}
%}
```

Listing 10-1 PIO Assembly Language code to blink a LED

Here are 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 it must follow the rules for a C variable.
- The PC wraps back to 0 once it passes 31 giving it an infinite loop for free. However, there are control registers that can alter this

wraparound, namely, setting the end instruction and 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 those in ARM Assembly Language, 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, C code can be included 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, and 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.

```
/**
 * 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;
}
```

Listing 10-2 The C code to call the SDK to download and configure the PIOs

The C program uses three PIO processors in PIO bank 0. 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.

```
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)
```

Listing 10-3 CMakeLists.txt file with the pico_generate_pio_header statement

The C code that calls SDK functions to control the PIOs is standard and taken from the various PIO samples included in the SDK. Next, the individual PIO instructions are looked at 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 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 allow waiting 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
```

OUT

Out transfers bits from the output shift register into various destinations inside the PIO. This data is received from the ARM CPU and has already been 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 **RxFIFO** 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

Pulls a 32-bit quantity from the Tx FIFO into the OSR. There are two parameters used to determine whether to block if the Tx FIFO is empty and what to do if the OSR isn't empty enough as prescribed by a configurable parameter. The non-blocking 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's often used to do this automatically, saving an instruction.

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.

Chapter 11 talks about interrupts, 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

Sets an immediate value to a destination. The immediate value is limited to 5 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 communications protocols. First, how to control the speed the program executes at is looked at.

About the Clock Divider

By default, each PIO instruction executes in one system clock cycle, unless it must wait on an external event. The system clock runs at 125MHz or 150MHz, 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 65,536 in increments of 1/256. The easiest way to configure this is via the Pico-series SDK function

```
static inline void sm_config_set_clkdiv(
          pio_sm_config *c, float div);
```

where the clock divider passes 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 the flashing LED program, the clock divider must be configured in the **blink_program_init** function from **blink.pio** as shown in Listing 10-4.

```
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);
}
```

Listing 10-4 The blink_program_init function setting the clock divider

Then it's called with

```
blink_program_init(pio, sm, offset, pin,
65536.0f);
```

Next, adjust the delay loops with

```
pio->txf[sm] = clock_get_hz(clk_sys) / freq /
65536;
```

Since the desired frequency is 5Hz, the delaying loop is reduced 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, there is fine control of how long each individual instruction executes.

About the Delay Operand

Each PIO instruction has 5 bits set aside for delay and side-setting. Side-set will be discussed shortly. In the meantime, all 5 bits are used for delay. The delay is specified in square brackets after the instruction and with 5 bits has values of 0–31, 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. This is combined with using the clock divider. The PIO Assembly Language code is shown in Listing 10-5.

```
.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!
```

Listing 10-5 PIO code to flash the LEDs without a delay loop

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 1 for why this value needs to be changed. Slowing the RP2040/RP2350 PIOs to something human readable is only barely possible; however, at computer-to-computer speeds, the techniques in this section are extremely powerful. Next, how to control the pins without using **SET** instructions is examined.

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 the PIO can be configured to make side-set optional. The downside is that this uses 1 bit of the 5 bits available to specify side-set or delay. Listing 10-6 contains the PIO Assembly Language to use 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!
```

Listing 10-6 PIO program to flash the LEDs using side-set

This program flashes twice as fast, since one of the delay bits is used 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 they can be configured to one FIFO of eight words, making the other 0.
2. **PUSH** and **PULL** instructions can often be eliminated 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 Pico-series. These are powerful processors for offloading communications functions from the two ARM CPU cores. PIO functionality was introduced and viewed in an example program to flash the LEDs. Next, all the instructions were looked at in detail, and then program timing was studied by modifying the flashing

LEDs program to use all the various techniques. Then side-set was looked at to control GPIO pins, and other useful configuration items were reviewed.

Chapter 11 looks at how to catch interrupts from internal and external devices and how to set interrupts from software.

Exercises

1. The system clock for an RP2040 is 125,000,000Hz. 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. How does that change for a 150MHz RP2350?
2. Using **side-set,** how fast can a square wave's frequency cycle?
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.
4. In the first example program in this chapter, remove the **SET** instruction by placing side-set on the **JMP** instructions.
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.

S. Smith, *RP2040 Assembly Language Programming*, Maker Innovations Series
https://doi.org/10.1007/979-8-8688-2202-5_11

11. How to Set and Catch Interrupts

Stephen Smith[1]
(1) Gibsons, BC, Canada

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 did 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, a timer is programmed, so when it goes off it interrupts the program to process the LEDs. This way a loop isn't needed, nor is the handling of the LEDs integrated into other parts of a larger program. This chapter looks at interrupts on the Pico-series, how they work, and how to put them to use.

In general, when handling I/O, data is often received randomly, and just a notification is needed when it is there to process. Interrupts provide a great way to do this. The ARM Cortex-M-series has powerful interrupt support that is well worth looking at. Before getting into the details, here is an overview of the Pico-series' interrupt mechanisms.

Overview of the Pico-series Interrupts

The ARM Cortex-M0+ powering the RP2040 supports 32 separate interrupt sources of which 26 are implemented, leaving 6 unused. The ARM Cortex-M33 powering the RP2350 supports 52 interrupts of which 46 are connected. Each of these interrupt sources wires an interrupt source, whether an internal or external device, to the 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.

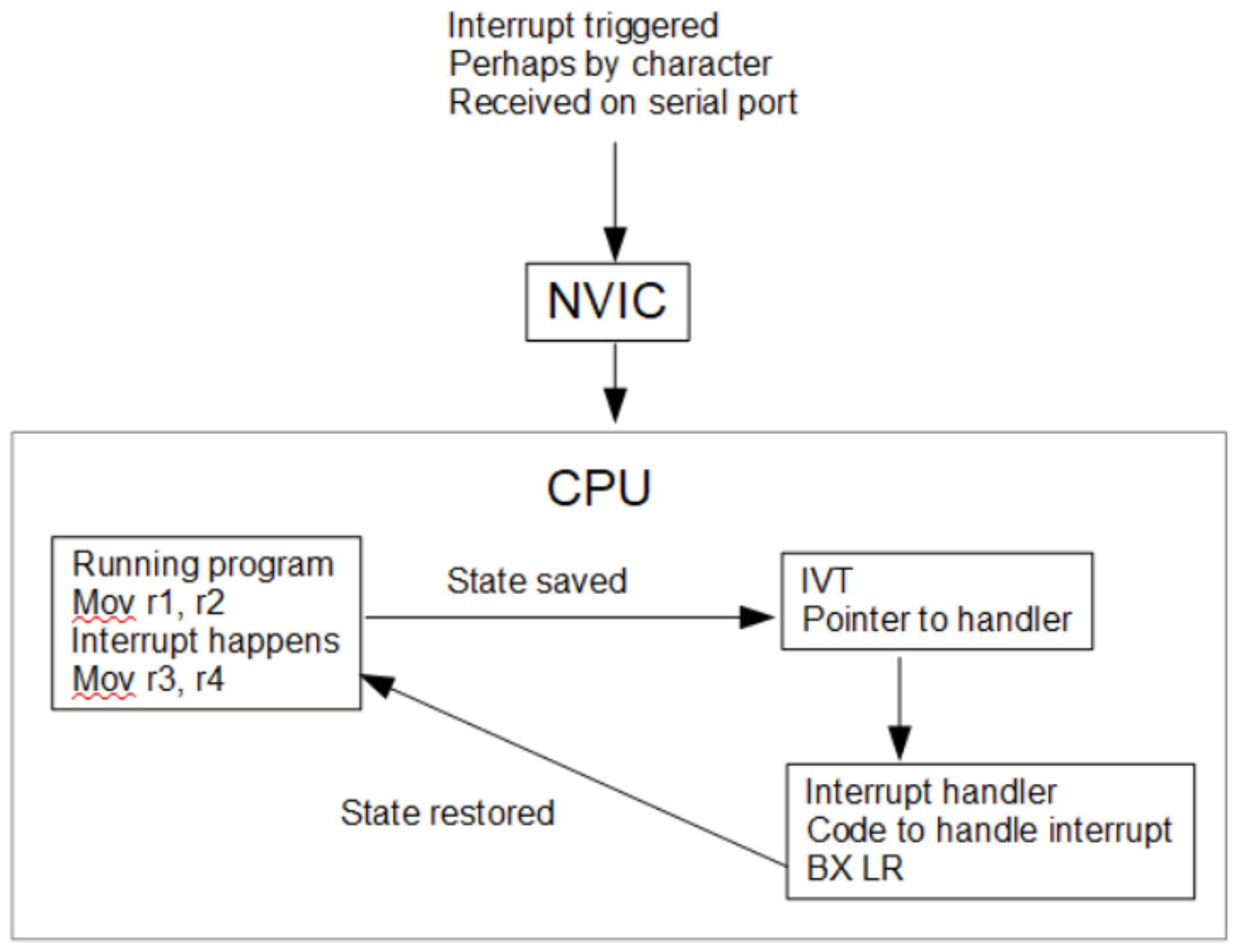


Figure 11-1 Overview of the interrupt calling process

With this overview in mind, the following sections dig into the various components in more detail starting with the list of interrupts.

About the RP2040 versus the RP2350

The RP2350 is based on the ARM Cortex-M33, whereas the RP2040 is based on the older ARM Cortex-M0+. The M33 supports more interrupts than the M0+, and consequently the RP2350 utilizes more interrupts and external devices. This section lists the values for the M33, but the final full program contains conditional compiles to work with either an RP2040 or RP2350. In the SDK if the board is a Pico 2, then the header file **m33.h** must be used, whereas if the board is a Pico 1, then **m0plus.h** must be included. Some of the defines in these files contain M33 or M0PLUS and must be conditionally compiled if both boards are supported.

Further, because the M33 now supports more than 32 interrupts, additional banks of hardware registers are required to control them; therefore, the constants contain 0 or 1 depending upon which bank needs to be used.

About the Pico-series' 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

Number	IRQ	Priority	Source	Comment
1	−15	−3	Reset	Triggered at power on or reset
2	−14	−2	NMI	Non-maskable interrupt
3	−13	−1	Hard fault	Triggered by non-recoverable hardware failures
4	−12	0	MemManage	Memory manager fault
5	−11	0	BusFault	Memory errors on the bus
6	−10	0	UsageFault	Usage error like undefined instruction
11	−5	0	SVCall	Triggered by the SVC instruction
12	−4	0	DebugMon	Debugger triggered
14	−2	0	PendSV	Triggered by the SVCall handler
15	−1	0	Systick	ARM system 24-bit clock tick

External interrupts are wired up to the CPU starting at IRQ0 but starting at exception 16. Both the IRQ number and exception number are used in various situations. In Table 11-1, the IRQs are negative, to show their relative place to the external interrupts. Exceptions 7–10 and 13 are unused and reserved for future use. This table is for the RP2350; some of these are not present in the RP2040. 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 some of the interrupts wired up to the ARM CPU inside the RP2350 SoC.

Table 11-2 A selection of the RP2350's interrupts and their priority

Number	IRQ	Priority	Source	Comment
16–19	0–3	2	Timer 0	Alarm 0
20–23	4–7	2	Timer 1	Alarm 1
24–25	8–9	2	PWM	Interrupt when a slice is complete
26–29	10–13	2	DMA	Direct Memory Access
30	14	2	USB	Data received
31–36	15–20	2	PIO	Programmable I/O
37–38	21–22	2	GPIO	One for each bank
39–40	23–24	2	QSPI	External flash memory
41–45	25–29	2	SIO	
46	30	2	Clocks	
47–48	31–32	2	SPI	Data received, data sent, buffer overrun
49–50	33–34	2	UART	11 possible reasons
51	35	2	ADC	FIFO reached threshold full
52–53	36–37	2	I2C	Data received or sent

Next, how the Pico-series assigns an interrupt handler for each of these is described.

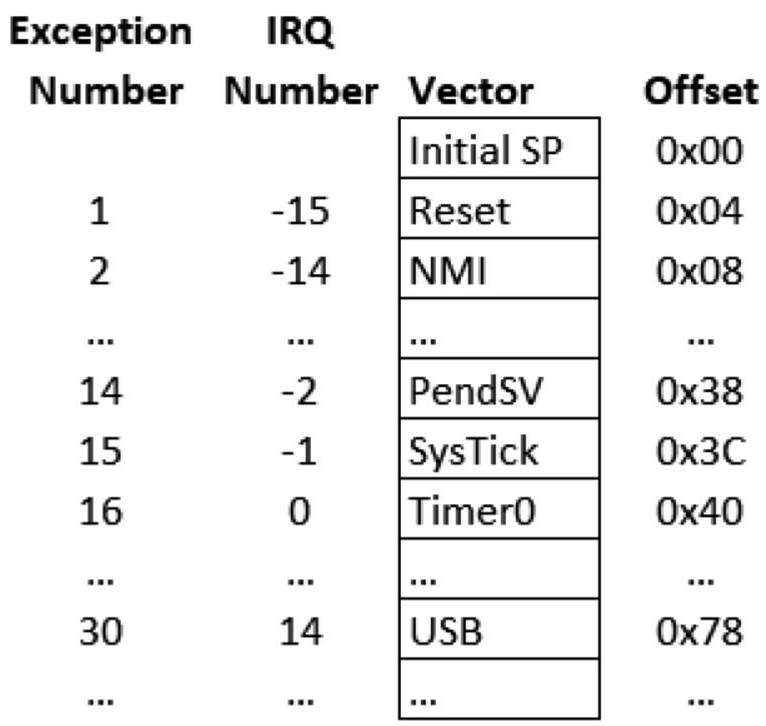
About the Interrupt Vector Table

When the Pico-series 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-M-series interrupt configuration. This table is a list of memory addresses, one for each interrupt. When an interrupt occurs, the ARM processor 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.



Exception Number	IRQ Number	Vector	Offset
		Initial SP	0x00
1	-15	Reset	0x04
2	-14	NMI	0x08
...	...	...	...
14	-2	PendSV	0x38
15	-1	SysTick	0x3C
16	0	Timer0	0x40
...	...	...	...
30	14	USB	0x78
...	...	...	...

Figure 11-2 Format of the Interrupt Vector Table

The easiest way to access the IVT is to read the hardware register where it's configured. **PPB_BASE** is defined for the memory address of the start of the ARM Cortex-M-series' hardware registers, and then **M33_VTOR_OFFSET** defined in **m33.h** is the offset to the IVT.

The value of **M33_VTOR_OFFSET** is too large to fit in an immediate operand, so it needs to be loaded from memory; then add these two numbers together to get the address of the hardware register containing 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/m33.h"
...
      LDR   R2, ppbbase
      LDR   R1, vtoroffset
      ADD   R2, R1
      LDR   R1, [R2]
...
ppbbase:    .word   PPB_BASE
vtoroffset: .word   M33_VTOR_OFFSET
```

Place the address of the interrupt handler into the correct offset within this table. When the Pico-series 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.

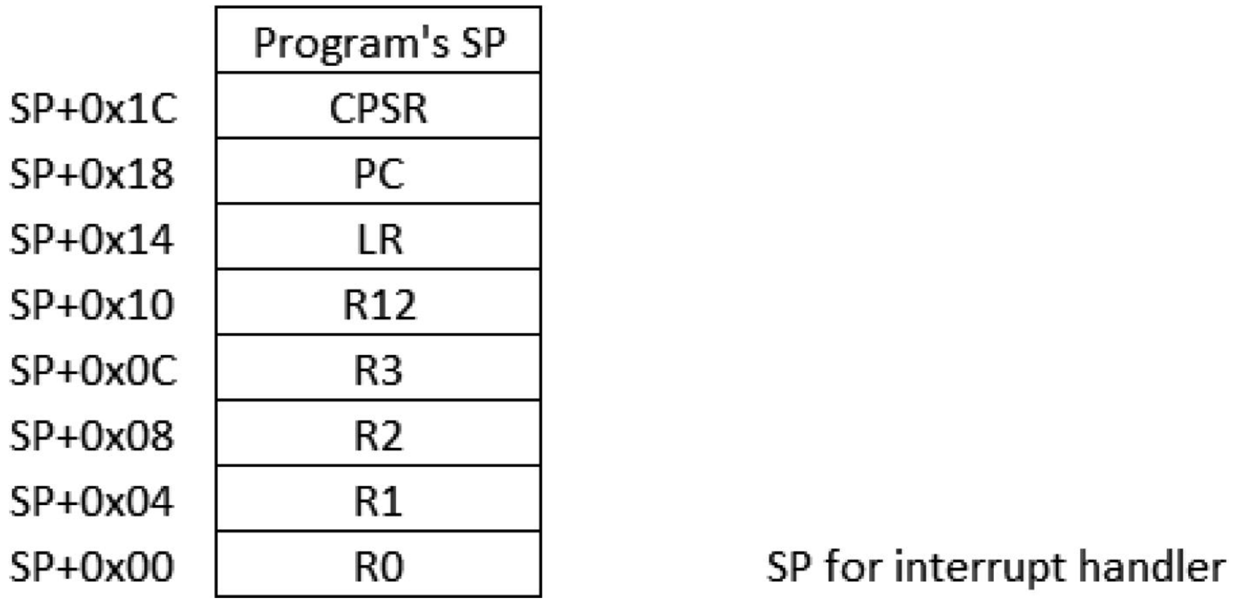


Figure 11-3 Processor's saved state while interrupt handler runs

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–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 Pico-series 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-M-series 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. Next, how this all fits together in a real application is looked at.

Flashing LEDs with Timer Interrupts

There are many techniques to flash three LEDs. Now this is done using the Pico-series' built-in timer via an interrupt. In this example, one of the Pico-series' alarms is set to interrupt the program every 200ms to switch to the next LED. The timer interrupt handler is implemented 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 pseudo-code 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
```

Listing 11-1 Pseudo-code for the alarm interrupt handler

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 Pico-series' alarm timer follows.

About the RP2040 Alarm Timer

The alarm timer is a 64-bit number that is incremented every microsecond. 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. So, in the code, the timer's count is read, 200,000 (200ms in microseconds) is added, and then the alarm is set.

The locations of the hardware registers are in **timer.h**, with the base address in **addressmap.h**. Below is the code to do this with the assumption **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 a timer interrupt is received, the interrupt must be cleared 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, it needs to set the interrupt handler and enable the timer IRQ with the NVIC.

Setting the Interrupt Handler and Enabling IRQ0

Previously, how to get the location of the IVT was learned, and in this program the interrupt handler is configured into it. Assuming the location of the IVT is in **R2**, then the interrupt handler is set 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 IRQ0 is enabled 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   M33_NVIC_ICPR0_OFFSET
setint:    .word   M33_NVIC_ISER0_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**.

```
@
@ Assembler program to flash three LEDs connected
to the
@ Raspberry Pico-series 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"
#if defined(PICO_RP2040)
#include "hardware/regs/m0plus.h"
#else
#include "hardware/regs/m33.h"
#endif

      .EQU LED_PIN1, 18
      .EQU LED_PIN2, 19
      .EQU LED_PIN3, 20

      .EQU FUNCSEL_VALUE_SIO, 5

      .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.
       MOV   R4, R0
       LSL   R4, #3                      @ each
GPIO has 8 bytes of registers
       LDR   R2, iobank0                 @ address
we want
       ADD   R2, R4                      @ add the
offset for the pin number
       MOV   R1, #FUNCSEL_VALUE_SIO
       STR   R1, [R2, #IO_BANK0_GPIO0_CTRL_OFFSET]
#if HAS_PADS_BANK0_ISOLATION
@ Remove pad isolation now that the correct
peripheral is set
       LDR   R2, padsbank0
       LSL   R3, R0, #2                  @ pin * 4
for register address
       ADD   R2, R3                      @ Actual
set of registers for pin
       LDR   R4, clearoffset
       ADD   R2, R4
       LDR   R1, PBGIB
       STR   R1, [R2, #PADS_BANK0_GPIO0_OFFSET]
#endif
       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
clearoffset: .word   REG_ALIAS_CLR_BITS
ppbbase:      .word   PPB_BASE
#if defined(PICO_RP2040)
timerbase:    .word   TIMER_BASE
vtoroffset:   .word   M0PLUS_VTOR_OFFSET
clearint:     .word   M0PLUS_NVIC_ICPR_OFFSET
setint:       .word   M0PLUS_NVIC_ISER_OFFSET
#else
timerbase:    .word   TIMER0_BASE
vtoroffset:   .word   M33_VTOR_OFFSET
clearint:     .word   M33_NVIC_ICPR0_OFFSET
setint:       .word   M33_NVIC_ISER0_OFFSET
PBGIB:        .word   PADS_BANK0_GPIO0_ISO_BITS
#endif
alarmtime:    .word   200000
printstr:     .asciz  "Couting %d\n"

.data
state:        .word   0
```

Listing 11-2 Flashing the LEDs via timer interrupts

There's nothing special about the CMakeLists.txt file; it just needs to compile **timeint.S**. Notice that everything was done using just registers **R0–R3**, so no other registers needed to be saved.

Note The **defined(PICO_RP2040)** is tested in the conditional compiles to determine which include files and constants to use.

That example used hardware interrupts. Now a note on 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

The SDK wasn't used so far, to provide a bare 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 to save some coding. 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 the architecture of the ARM Cortex-M-series interrupt system was studied, including how to set an interrupt handler and enable and configure interrupts, as well as how state is saved and how interrupts can be interrupted in a nested manner.

The Pico-series' timer device was looked at next in detail including how to use it to set alarms to interrupt the program on a regular basis. A complete program was examined to demonstrate all these concepts in action, again while flashing the three LEDs. Then software-triggered service interrupts were shown, and the Pico-series SDK support was mentioned.

So far only the addition and subtraction of integers were covered. Chapter 12 covers a much broader set of mathematical operations.

Exercises

1. Most software engineers work hard to make their interrupt handlers operate 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 and why?
2. Debugging the program shows that the IVT is at the start of SRAM at memory location 0x20000000. Why not hard-code that in the program and save a couple of instructions?
3. Modify the state machine in the sample program to create a pattern where two LEDs are lit at the same time.
4. Implement the sample program in C using the SDK.
5. Create a small Assembly Language program to use the SVC instruction and handle the interrupt, printing something to know that it was triggered.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
S. Smith, *RP2040 Assembly Language Programming*, Maker Innovations Series
https://doi.org/10.1007/979-8-8688-2202-5_12

12. Multiplication, Division, and Floating Point

Stephen Smith[1]
(1) Gibsons, BC, Canada

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. How to perform more advanced mathematical functions is now explained.

Modern microcontrollers are quite sophisticated devices, but to keep costs low, they have far fewer transistors than full desktop CPUs like the ARM A-series. As a result, they lack the full floating-point units and sophisticated mathematics that their big brothers support. When a vendor such as Raspberry licenses an ARM M-series CPU, they can select from a number of options to include or exclude depending on their price target and desired transistor count.

The built-in and optional mathematical components included with the ARM Cortex-M0+ included in the RP2040 and the M33 included in the RP2350 are quite different. Table 12-1 summarizes these differences.

Table 12-1 Comparison of the mathematical capabilities of the RP2040 versus RP2350

Operation	RP2040	RP2350
Integer multiplication	Included	Included
Integer division	Coprocessor	Included
DSP (Digital Signal Processor)-type function	Coprocessor	Coprocessor
Single-precision floating point	Boot-ROM library	FPU
Double-precision floating point	GCC library	Partial-FPU

This chapter covers the abilities of the RP2350, since it makes sense to use this processor if these are needed. For instance, to perform a single-precision floating-point instruction using the library in the boot ROM typically takes 70 cycles, whereas the FPU in the RP2350 can perform this in two to three cycles.

Multiplication

Integer 32-bit multiplication is built into the ARM Cortex-M-series, 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 2), since the difference is in the discarded upper 32 bits. Here are 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 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. The M33 has division built in, so the RP2350 has a division instruction. The division instructions are

- SDIV {Rd}, Rn, Rm
- UDIV {Rd}, Rn, Rm

where

- **Rd**: Is the destination register
- **Rn**: Is the register holding the numerator
- **Rm**: Is the register holding the denominator

There are a few problems or technical notes on these instructions:

- There is no "S" option of this instruction, as it doesn't set the CPSR at all.
- Dividing by 0 should throw an exception; with these instructions it returns 0, which can be very misleading.
- The instruction only returns the quotient, not the remainder. Many algorithms require the remainder, and this must be calculated as remainder = numerator – (quotient * denominator).

The code to execute the division instructions is simple as follows:

```
MOV    R2, #100
MOV    R3, #4
SDIV   R4, R2, R3
UDIV   R4, R2, R3
```

Interpolation

Both the RP2040 and RP2350 have 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 Pico-series 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 four-dollar chip.

DSPs typically perform full floating-point computations, contain instructions that are helpful for processing input signals, and have their own instruction sets. The Pico-series interpolators can assist with some of the same algorithms as full DSP chips but still rely on the ARM Cortex-M-series to do much of the work. The interpolators contain their own registers and perform addition, multiplication, and some bit operations. They're 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, piece by piece is built up starting with the simplest example of adding some integers.

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)]
```

The .**EQU** directive will be used for each of these, to keep the length of each line down. The first and easiest example is next.

Adding an Array of Integers

To get used to working with the interpolator, first of all 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, the lane control register **SIO_INTERP0_CTRL_LANE0** is configured 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 0. At the end, the value from accumulator 0 is read for the final result. Listing 12-1 shows the Assembly Language code to perform this.

```
.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]
```

Listing 12-1 Using one of the interpolators to add an array of integers

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. Here's 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, 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 results in 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 12-2. 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, and then the multiplication isn't as accurate as a full floating-point calculation but is much faster.

```
.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]
```

Listing 12-2 Code to interpolate between some numbers and keep the sum of the results

Lane 0 is configured for blend mode and raw add mode. The necessary bit pattern could've been derived and done this in fewer instructions, but since this is initialization code, it was left separate for readability.

Lane 0 needed to be configured to not mask any bits; the configuration is to allow bits 0 to bits 31 through, which is what is needed in this case, see Exercise 3.

To read the result registers, read either the **PEEK** or **POP** register. **PEEK** reads the result without doing anything else. **POP** reads the value, but 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 *RP2350 Datasheet* has a complete reference of all the functionality, and the Pico-series

SDK samples have a good selection of algorithms making use of the interpolator. How to use floating-point numbers and arithmetic from the Assembly Language programs is covered next.

Floating Point

The RP2040 doesn't have floating-point hardware and relies on optimized routines contained in the Raspberry Pico 1 boot ROM. The M33 contained in the RP2350 contains a single-precision floating-point FPU, greatly speeding and simplifying floating-point calculations.

Defining Floating-Point Numbers

The GNU Assembler has directives for defining storage for both single- and double-precision floating-point numbers. These are **.single** and **.double**, for example:

```
.single   1.343, 4.343e20,   -0.4343, -0.4444e-10
.double  -4.24322322332e-10,  3.141592653589793
```

These directives always take base 10 numbers. The RP2350 only supports single-precision floating point, but there is support for double precision in a separate coprocessor.

About Floating-Point Registers

The ARM FPU has its own set of registers. There are 32 single-precision floating-point registers that are referred to as S0, S1, ..., S31.

Chapter 7 gave the protocol for who saves which registers when calling functions. These floating-point registers need to be added to the protocol.

- **Callee saved**: The function is responsible for saving registers **S16–S31**, if the function uses them.
- **Caller saved**: Registers **S0–S15** must be saved by the caller if they are required to be preserved.

There are FPU-specific functions to access these registers, for instance, to push and pop these to and from the stack:

```
VPUSH   {reglist}
VPOP   {reglist}
```

For example:

```
VPUSH   {S16-S31}
VPOP   {S16-S31}
```

Loading and Saving FPU Registers

In Chapter 6, the **LDR** and **STR** instructions were covered to load registers from memory and then store them back to memory. The floating-point coprocessor has similar instructions for its registers:

```
VLDR   Sd, [Rn{, #offset}]
VSTR   Sd, [Rn{, #offset}]
```

Both instructions support pre-index addressing offsets, for example:

```
      LDR    R1, =fp1
      VLDR   S4, [R1]
      VLDR   S4, [R1, #4]
      VSTR   S4, [R1]
      VSTR   S4, [R1, #4]
      ...
.data
fp1: .single 3.14159
fp2: .single 4.3341
fp3: .single 0.0
```

Basic Arithmetic

The floating-point processor includes the four basic arithmetic operations, along with a few extensions like square root.

Here is a selection of the instructions:

```
VADD.F32   {Sd}, Sn, Sm
VSUB.F32   {Sd}, Sn, Sm
VMUL.F32   {Sd,} Sn, Sm
```

```
VDIV.F32   {Sd}, Sn, Sm
VSQRT.F32  Sd, Sm
```

If the destination register is in curly brackets {}, it is optional so it can be left out. This means apply the second operand to the first, so to add **S1** to **S4**, simply write

```
VADD.F32   S4, S1
```

These functions are all fairly simple, so next is an example to show many of these in use.

Sample Floating-Point Program

Given two points (x_1, y_1) and (x_2, y_2), then the distance between them is given by the formula

$d = \text{sqrt}((y_2-y_1)^2 + (x_2-x_1)^2)$

Listing 12-3 is a function to calculate this for any two single-precision floating-point pair of coordinates. Place this function in the file **distance.S**.

```
@
@ Example function to calculate the distance
@ between two points in single precision
@ floating point.
@
@ Inputs:
@       R0 - pointer to the 4 FP numbers
@               they are x1, y1, x2, y2
@ Outputs:
@       R0 - the length (as single precision FP)

.global distance      @ Allow function to be
called by others

@
distance:
        @ push registers that need to be saved
        push   {LR}

        @ load all 4 numbers at once
        vldm   R0, {S0-S3}

        @ calc s4 = x2 - x1
        vsub.f32    S4, S2, S0
        @ calc s5 = y2 - y1
        vsub.f32    S5, S3, S1
        @ calc s4 = S4 * S4 (x2-X1)^2
        vmul.f32    S4, S4
        @ calc s5 = s5 * s5 (Y2-Y1)^2
        vmul.f32    S5, S5
        @ calc S4 = S4 + S5
        vadd.f32    S4, S5
        @ calc sqrt(S4)
        vsqrt.f32   S4, S4
        @ move result to R0 to be returned
        vmov        R0, S4

        @ restore what we preserved.
        pop   {PC}
```

Listing 12-3 Function to calculate the distance between two points

Now place the code from Listing 12-4 in **floatingpoint.S**, which calls distance three times with three different points and prints out the distance for each one.

```
@
@ Examples of the floating point routines.
@

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

      .equ  N, 3               @ Number of points.
```

```
main: BL    stdio_init_all   @ initialize uart or
usb
      ldr   r6, =points      @ pointer to current
points
      mov   r7, #N           @ number of loop
iterations

loop: mov   r0, r6           @ move pointer to
parameter 1 (r0)
      bl    distance         @ call distance
function

@ need to take the single precision return value
@ and convert it to a double, because the C printf
@ function can only print doubles.
      bl    __aeabi_f2d
      mov   r2, r0
      mov   r3, r1
      ldr   r0, =prtstr      @ load print string
      bl    printf           @ print the distance

      add   r6, #(4*4)       @ 4 points each 4
bytes
      sub   r7, #1           @ decrement loop
counter
      cmp   r7, #0           @ is the loop done?
      bne   loop             @ loop if more points

loop2:
      B   loop2

.data
        .align         4      @ necessary
alignment
points: .single      0.0,     0.0,   3.0,      4.0
        .single      1.3,     5.4,   3.1,     -1.5
        .single 1.323e10, -1.2e-4, 34.55, 5454.234
prtstr: .asciz "Distance = %f\n"
```

Listing 12-4 Main program to call the distance function three times

Some Notes on C and printf

Besides the usage of the FPU instructions like **vmul.f32**, there is a call to **__aeabi_f2d** to convert the 32-bit floating-point number for the distance to a 64-bit number. 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's no way to pass a single-precision 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 uses two 32-bit registers, if they are available or are placed 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 **R1** is not used in calls to **printf**. A 64-bit quantity can be returned in registers **R0** and **R1**, which is in the code.

The FPU integrated into the RP2350 doesn't contain any conversion routines, so these must be performed in software outside of the FPU. In this case a routine from the Pico-series SDK is used.

Summary

In this chapter, the integer multiplication and division instruction were studied. Next, the Pico-series interpolator coprocessor and how to use it to interpolate as well as perform multiply and accumulate operations were covered. The interpolator also has some bit manipulation operations that combine to give limited DSP-like capabilities for input data processing.

The RP2350 contains a single-precision floating-point unit; the registers and basic instructions for floating-point operations were presented. An example to calculate the distance between two points was then examined to see how to use all these instructions together. An aside on converting single- to double-precision numbers was presented to explain how to use **printf** to see the results.

So far in this book everything was done on one of the two ARM Cortex-M-series CPU cores. In Chapter 13, how to use the second CPU core and coordinate the work between the two CPUs is explained.

Exercises

1. Create a small program using the multiplication and division examples and single step through it in the debugger to ensure how it works is understood.
2. Examine the bits of calculating –1 * 4 to see why it works either interpreting these as unsigned or signed integers.
3. In the interpolation example, lane 1 was set to the value 0x7c00. Look up the definition of the bits for the lane control register in the *RP2350 Datasheet* and see how this allows all the bits through with no masking.
4. The area of a circle is π * r^2. Write a small Assembly Language program that uses the FPU floating-point routines to calculate the area of circles with radii 1, 1.4, and 3. Print out the results.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
S. Smith, *RP2040 Assembly Language Programming*, Maker Innovations Series
https://doi.org/10.1007/979-8-8688-2202-5_13

13. Multiprocessing

Stephen Smith[1]
(1) Gibsons, BC, Canada

The Pico-series contains two ARM Cortex-M-series CPU cores. This chapter looks at how to run code on the second processor. The second processor is in power-conserving sleep state by default; how to wake it up and assign it work to process will be shown. The Raspberry company added the following helpful features to the Pico-series for working with both CPU cores:

1. There are two First-In-First-Outs (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.

3. The RP2350 adds a doorbell interrupt where one core can interrupt the other core.

These 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. First, become familiar 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 Pico-series chip.

Since most applications don't use the second CPU, it's 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 an 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. This command 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 Pico-series 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 and 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 Pico-series 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 use the following code:

```
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 whether the FIFO

1. Contains data
2. Is full
3. Was read when empty
4. Was written to when it was full, so it was discarded

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.

```
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
```

Listing 13-1 Interprocessor FIFO read routine

This routine is blocking; if there's 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, how to get code running on the core 1 CPU is looked at.

How to Run Code on the Second CPU

When the Pico-series is powered on, both CPU cores receive a RESET interrupt and the initial 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's 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's 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 the one for core 0)
4	SP	Top of stack (stack grows down)
5	routine	Thumb routine to run (address must be odd)

The code that follows provides the same IVT as core 0, 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 the top of the stack is passed in 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 or **arm8_bootrom_rt0.S** for the RP2350 in their respective **bootrom github** repositories.

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 of all, read the following 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 ($\Phi = (1 + \sqrt{5}) / 2 = \sim1.618$), which is also the limit of the ratio of consecutive Fibonacci numbers as n goes to infinity.

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 the first few of these can be calculated. The first few factorials are

```
1, 2, 6, 24, 120, 720, 5040, 40320, ...
```

Factorials are common in probability and combinatorics. With these in hand, the complete program is presented.

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.

```
@
@ Example using the second core for processing.
@

#include "hardware/regs/addressmap.h"
#if defined(PICO_RP2040)
#include "hardware/regs/m0plus.h"
#else
#include "hardware/regs/m33.h"
#endif
#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
#if defined(PICO_RP2040)
vtoroffset: .word   M0PLUS_VTOR_OFFSET
#else
vtoroffset: .word   M33_VTOR_OFFSET
#endif
```

```
@ 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
```

Listing 13-2 Multiprocessor program to calculate Fibonacci numbers and factorials

The routines that calculate Fibonacci numbers and factorials are straightforward, implementing a simple FOR loop to calculate the desired number. It's worth reviewing these to ensure understanding of 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's empty. The SDK warns that there could be leftover 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's 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 2 to implement blocking.
3. **fifo_pop**: Checks the status register to see if there's data available; if there isn't, it goes to sleep by issuing a **WFE** instruction and loops back. If there's 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 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 Pico-series SDK 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 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 that the two processors don't interfere with each other. For instance, if both processors update a table in memory, it isn't desirable if one processor overwrites the work of the other. When this goes wrong, this leads to hard-to-replicate bugs that are difficult to find.

The Pico-series provides 32 spinlocks to regulate access to shared resources. A spinlock is a resource that a CPU tries to acquire, but if the other CPU 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 the SDK isn't being used, the program chooses 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 it's been successfully acquired; however, if the value read is zero, spin the program to wait to acquire it. Listing 13-3 shows the 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
```

Listing 13-3 Code to lock a spinlock

To release a spinlock, any value is written to the spinlock's hardware register. Listing 13-4 shows the code to release a spinlock.

```
LDR   R1, spinbase
STR   R0, [R1]      @ value written doesn't matter
```

Listing 13-4 Code to unlock a spinlock

Next is 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 0–99 and their squares. It also puts the core number in each row, to mark the row as done, so which core filled in each row can be seen. If spinlocks weren't used, then the cores would overwrite each other’s work. Even though a row is marked as used first, there’s a window of opportunity where both cores read a row as available, 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 that 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.

```
@
@ Example using the second core for processing.
@ Protecting a memory table with a spin lock.
@

#include "hardware/regs/addressmap.h"
#if defined(PICO_RP2040)
#include "hardware/regs/m0plus.h"
#else
#include "hardware/regs/m33.h"
#endif
#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
#if defined(PICO_RP2040)
vtoroffset:  .word   M0PLUS_VTOR_OFFSET
#else
vtoroffset:  .word   M33_VTOR_OFFSET
#endif

@ Spinlock 24 is first one available for exclusive
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
```

Listing 13-5 Program to update the table of squares using both cores

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 4.

In the main program after starting core 1 and filling in its 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 5.

In this chapter code was written directly to the hardware registers; however, there are Pico-series SDK functions that can be used as follows.

A Word on the SDK

The Pico-series 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 Pico-series works and provide intuition-based instructions for a deeper knowledge of how the operations work.

Summary

This chapter covered how to use the second CPU core contained on the RP2040 or RP2350. 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 were explained. Since both CPU cores access the same memory on the Pico-series, 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 a Pico-series microcontroller to the world wide web is covered.

Exercises

1. Add error checking to **launch_core1**. Break out the sending and receiving of 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.
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; enter a low-power state while waiting, like how **fifo_pop** waits for data to arrive.
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, have core 0 read it, and then use the result, in this case, to print it. Change the program to work this way, so core 1 is purely a computation service that's called to calculate factorials.
4. Remove the ten NOP instructions after the table row is written. How does that affect the results? Explain what's going on. How can few NOPs maintain an even workload?
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.
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 understand how they work.
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.

S. Smith, *RP2040 Assembly Language Programming*, Maker Innovations Series
https://doi.org/10.1007/979-8-8688-2202-5_14

14. How to Connect Pico to IoT

Stephen Smith[1]
(1) Gibsons, BC, Canada

This chapter presents a complete realistic microcontroller project written entirely in Assembly Language. A Pico-series device collects data and then provides it to a central server. Since this is a book on Assembly Language and not electronics, components built into the Pico-series are used, rather than requiring extra components. The built-in temperature sensor is used to collect data, and then the program communicates with a server using UART1. It's used rather than UART0, so that UART0 can be used for debugging and receive output from **printf** statements. The assumption is that a Raspberry Pi is used for debugging and development, so this is used as the server and a Python program is written to poll the various devices connected to it for data.

The Raspberry Pi 5's UART is connected with TX on pin 8 and RX on pin 10. For the Raspberry Pi Pico-series, UART1 is connected to GPIO 4 and GPIO 5. This makes physical pin 6 TX and pin 7 RX. Connect the RX from the Raspberry Pi 5 to the TX on the Pico-series and the TX to the RX.

This project gives an 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 of all, the Pico-series analog-to-digital converter (ADC) and the built-in temperature sensor are presented.

About the Pico-series 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 Pico-series built-in one, measure the voltage of a biased bipolar diode, which varies depending on the ambient temperature. The Pico-series datasheet then provides a formula to convert this voltage to temperature.

The Pico-series 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 0–3.3V, so to convert from the 12-bit number to voltage, multiply it by $3.3/2^{12}$. The *RP2350 Datasheet* gives a formula to convert this voltage into degrees Celsius.

Doing it this way requires floating-point arithmetic, which isn’t preferred. Instead, combine these two formulas—see Exercise 4—to derive a formula that can be evaluated easily using only integer arithmetic:

```
Temp = 437 - (100 * rawADC) / 215
```

To divide the **rawADC** by 2.15, multiply both the numerator and denominator by 100, which is a good trick to only use integer arithmetic. This is performed in the **calcTempCelc** function that uses the **SDIV** instruction on an RP2350 or the division coprocessor on an RP2040.

The **ADC** has a status and control register that enables both the **ADC** and the temperature sensor, although these are turned off by default to save power. The **ADC** connects to four GPIO pins numbered 0–3 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 finish powering up before its ready for use. Similarly, when a temperature reading is taken, 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**.

```
@
@ Module to interface to the RPxxxx 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
celcius.
@ Calculates degrees = 437 - 100 * R0 / 215
@
```

```
@ Registers:
@ Input:   R0 - raw 12-bit ADC value
@ Output:  R0 - degrees celcius
@ 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
#if defined(PICO_RP2040)
          BL    intDivide                    @ R0
= R0 / 215
#else
          SDIV  R0, R1
#endif
          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 to complete
          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
```

Listing 14-1 Routines to activate the **ADC** controller and read the temperature

This chapter separates the various functions into separate source code modules, to describe 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 Communications Protocol

In this simple setup, the Pico-series 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 following 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 5—that simply demonstrates the start of a more full-featured protocol. Each device connected to the twisted wire pair 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 a **SOH** character if it isn’t. The program polls a range of addresses and has a one-second timeout, 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 that should be stored in a file called **serpolling.py** and run from the Thonny Python IDE.

```
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',
        port = '/dev/ttyAMA0',
        baudrate = 115200,
        timeout=1
        )

while 1:
    for addr in range(49, 53):
        msg = sendPollreadResp(addr)
        print( msg )
        time.sleep(1)
```

Listing 14-2 The Python server program

Note The serial port's device name has changed from Raspberry Pi OS version to version. At the time of writing, it is **/dev/ttyAMA0** on Bookworm; previously it was **/dev/serial0**. On Trixie it is **/dev/ttyACM0**. It might change again in the future.

With the server polling done, now back to the Pico-series microcontroller to see how to use the UART to receive the poll and respond to it.

About the Pico-series UART

The UART device on the RP2040/RP2350 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 one of the two built-in UART controllers is used. Like all connected hardware, there is a bank of hardware registers for controlling these. There are two registers for setting the baud rate and the speed at which 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-series doesn't have a way to connect any of these to external GPIO pins, so a lot of the UART controller's 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 *RP2350 Datasheet* and left to Exercise 8 for the general case.

The line control register **UARTLCR_H** sets

1. On the 8-bit mode, by setting the two WLEN bits to 1
2. The FEN bit which enables the FIFOs

Parity is not enabled, so it stays off.

The control register **UARTCR** sets the bits to

1. Enable the receiver
2. Enable the transmitter
3. Enable the UART

When reading a byte, the flag register **UARTFR** is used 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.

These conditions are busy-waited on in the routines in Listing 14-3 that goes in a file called **muart.S**.

```
@
```

```
@ 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 UART1.
@ 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, uart1base
       @ Set baud rate to 115200
       @ See the RP2040 datasheet for the magic
values 67 and 52
#if defined(PICO_RP2040)
       MOV   R0, #67
       STR   R0, [R1, #UART_UARTIBRD_OFFSET]
       MOV   R0, #52
       STR   R0, [R1, #UART_UARTFBRD_OFFSET]
#else
       MOV   R0, #81
       STR   R0, [R1, #UART_UARTIBRD_OFFSET]
       MOV   R0, #24
       STR   R0, [R1, #UART_UARTFBRD_OFFSET]
#endif
       @ 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, #4                          @
GPIO4 pin is UART1 TX
       BL    gpioInit
       MOV   R0, #5                          @
GPIO5 pin is UART1 RX
       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, uart1base                     @
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 rx fifo empty
       AND   R0, R2
       BNE   waitr                             @
set means fifo empty
```

```
       @ Read the character
       LDR   R0, [R1, #UART_UARTDR_OFFSET]     @
read the received 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, uart1base
       @ 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
       MOV   R8, R0                          @
Save pin number
       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_GPIO4_CTRL_FUNCSEL_VALUE_UART1_TX
       STR   R1, [R2, #IO_BANK0_GPIO0_CTRL_OFFSET]
#if HAS_PADS_BANK0_ISOLATION
@ Remove pad isolation now that the correct
peripheral is set
       LDR   R2, padsbank0
       MOV   R0, R8                           @
restore pin numbere
       LSL   R3, R0, #2                       @
pin * 4 for register address
       ADD   R2, R3                           @
Actual set of registers for pin
       LDR   R4, clearoffset
       ADD   R2, R4
       LDR   R1, PBGIB
       STR   R1, [R2, #PADS_BANK0_GPIO0_OFFSET]
#endif
       BX    LR

             .align  4
uart1base:   .word   UART1_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
clearoffset:  .word   REG_ALIAS_CLR_BITS
#if HAS_PADS_BANK0_ISOLATION
PBGIB:        .word   PADS_BANK0_GPIO0_ISO_BITS
#endif
```

Listing 14-3 The module for controlling serial communications

> **Note** The baud rate division constants are conditionally compiled since the RP2040 runs at 125MHz and the RP2350 at 150MHz.
>
> The RP2350 requires the extra step of enabling the pads where the code is conditionally compiled in, if needed.

Now that characters can be received and transmitted over the serial connection, a utility math routine is required.

Converting Integers to ASCII

A routine is needed to convert binary integers into ASCII strings. This is done backward, by first of all getting the least significant digit and next the most significant digit last and then reversing the digits at the end. This is done by repeatedly dividing by ten. 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; then negate the number to make it positive. The algorithm works for negative numbers, except for 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 on the RP2040 are in Listing 14-4 that should go in a file called **mmath.S**.

```
@
@ 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 cyles,
@ 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

#if defined(PICO_RP2040)
@ 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
```

```
#endif

@ 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
#if defined(PICO_RP2040)
         BL   intDivide
#else
         MOV   R2, R0                 @ Keep to calc
remainder
         SDIV  R0, R1                 @ R0 is quotient
         MOV   R7, R0
         MUL   R7, R1
         SUB   R1, R2, R7
#endif
         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}
```

Listing 14-4 Routines for division and converting integers to ASCII

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**.

```
@
@ 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 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:           "
```

Listing 14-5 The main driving program

The **CMakeLists.txt** file for this project is presented in Listing 14-6.

```
cmake_minimum_required(VERSION 3.13)

set(PICO_BOARD pico2 CACHE STRING "Board type")

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 1)
pico_enable_stdio_usb(iot 0)

pico_add_extra_outputs(iot)

target_link_libraries(iot pico_stdlib)
```

Listing 14-6 CMakeLists.txt file for this project

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

The Internet of Things (IoT) often refers to connecting microcontrollers to the Internet directly. There are wireless versions of both the Pico 1 and Pico 2, the W versions, which add Wi-Fi and Bluetooth. These add a standard radio module to either board, but unfortunately these modules are proprietary and don't document their interfaces publicly. To use these, the vendor's supplied SDK needs to be used, which is integrated into the Pico-series SDK. There are plenty of examples of using the SDK to communicate with the Internet. The easiest and best support is via MicroPython.

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 Pico-series SDK functions. Although Assembly Language is typically used to code highly specialized functions that 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.

At this point, it should be clear how to write Assembly Language code for Pico-series chips. The fundamentals of writing basic programs and interfacing with hardware integrated into the Pico-series were covered.

Now go forth and experiment. The only way to learn programming is by doing. Think up some Assembly Language projects. The RP2040 and RP2350 are flexible devices that can interface to nearly anything including any sensor or device that can be connected to the Arduino and Raspberry Pi systems.

Exercises

1. Change the program to report in degrees Fahrenheit rather than degrees Celsius.
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

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.

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 a SOH character. Why is this a good practice?
4. Combine the formula for converting raw **ADC** to voltage with the temperature formula in the *RP2350 Datasheet* to derive the temperature formula.
5. The simple protocol has no error checking. One technique is to add an XOR checksum to the message. Simply XOR all the bytes of the message together and include the checksum before the ETX character. Implement this for the protocol. How to best ensure the checksum isn’t one of the three special protocol characters?
6. The simple protocol has no authentication. Should a terminal need to supply authentication information? What are the pros and cons of adding this?
7. Typical temperatures are around room temperature or 20°C, two digits positive. Set up 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?
8. In the **initUART** function, the baud rate is hard-coded to 115,200. Change the routine to take the baud rate as a parameter and perform the calculations explained in the *RP2350 Datasheet* to configure the two baud rate registers correctly.

Note The calculation must consider the difference in the RP2040 versus RP2350 clock speed.

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 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
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

Dec	**Hex**	**Char**	**Description**
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
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
56	38	8	Eight
57	39	9	Nine

Dec	Hex	Char	Description
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
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

Dec	Hex	Char	Description
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
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

Dec	Hex	Char	Description
122	7A	z	Small z
123	7B	{	Left curly bracket
124	7C	|	Vertical bar
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	
148	94	ö	
149	95	ò	
150	96	û	
151	97	ù	
152	98	ÿ	
153	99	Ö	

Dec	Hex	Char	Description
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	¬	
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	╕	

Dec	Hex	Char	Description
185	B9	╣	
186	BA	║	
187	BB	╗	
188	BC	╝	
189	BD	╜	
190	BE	╛	
191	BF	┐	
192	C0	└	
193	C1	┴	
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	╫	

Dec	Hex	Char	Description
216	D8	╪	
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	∩	
240	F0	≡	
241	F1	±	
242	F2	≥	
243	F3	≤	
244	F4	⌠	
245	F5	⌡	
246	F6	÷	
247	F7	≈	

Dec	Hex	Char	Description
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	Pad 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-precision 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.
.fill	Define and fill some memory.
.float	Define memory for single-precision floating-point data.
.global	Make a symbol global, needed if reference from other files.
.hword	Defines memory for 16-bit integers.
.if	Marks the beginning of code to be conditionally assembled.
.include	Merge a file into the current file.
.int	Define storage for 32-bit integers.
.long	Define storage for 32-bit integers (same as .int).
.macro	Define a macro.
.octa	Defines storage for 64-bit integers.
.quad	Same as .octa.
.rept	Repeat a block of code multiple times.
.set	Set the value of a symbol to an expression.

Directive	Description
.short	Same as .hword.
.single	Same as .float.
.text	Generate following instructions into the code section.
.word	Same as .int.

Appendix C Binary Formats

This appendix describes the basic characteristics of the data types used in this book.

Integers

The following table provides the basic integer data types used. Signed integers are represented in the 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

Floating Point

The RP2040/RP2350 floating-point routines use the IEEE-754 standard for representing floating-point numbers. All floating-point numbers are signed.

Addresses

All addresses or pointers are 32-bit.

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

Size	**Range**	**C Type**
64	2.22507385850720138e-308 to 1.79769313486231571e+308	double

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-M-series 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
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

Instruction	Description
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
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

Appendix E Answers to Exercises

This appendix has answers to selected exercises. For program code, check the online source code at the Apress GitHub site.

Chapter 2

```
1. 0100 1101 0010, 0x4d2
```

Chapter 4

1. 177 (0xb1), 233 (0xe9)
2. -14, -125
3. 0x78563412
4. 0x118
5. 0x218

Chapter 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.

Chapter 9

1. 0x40044000, i2c.h
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

```
1. 65104, 78,125
2. 62,500,000Hz or 62.5MHz on an RP2040
```

Index