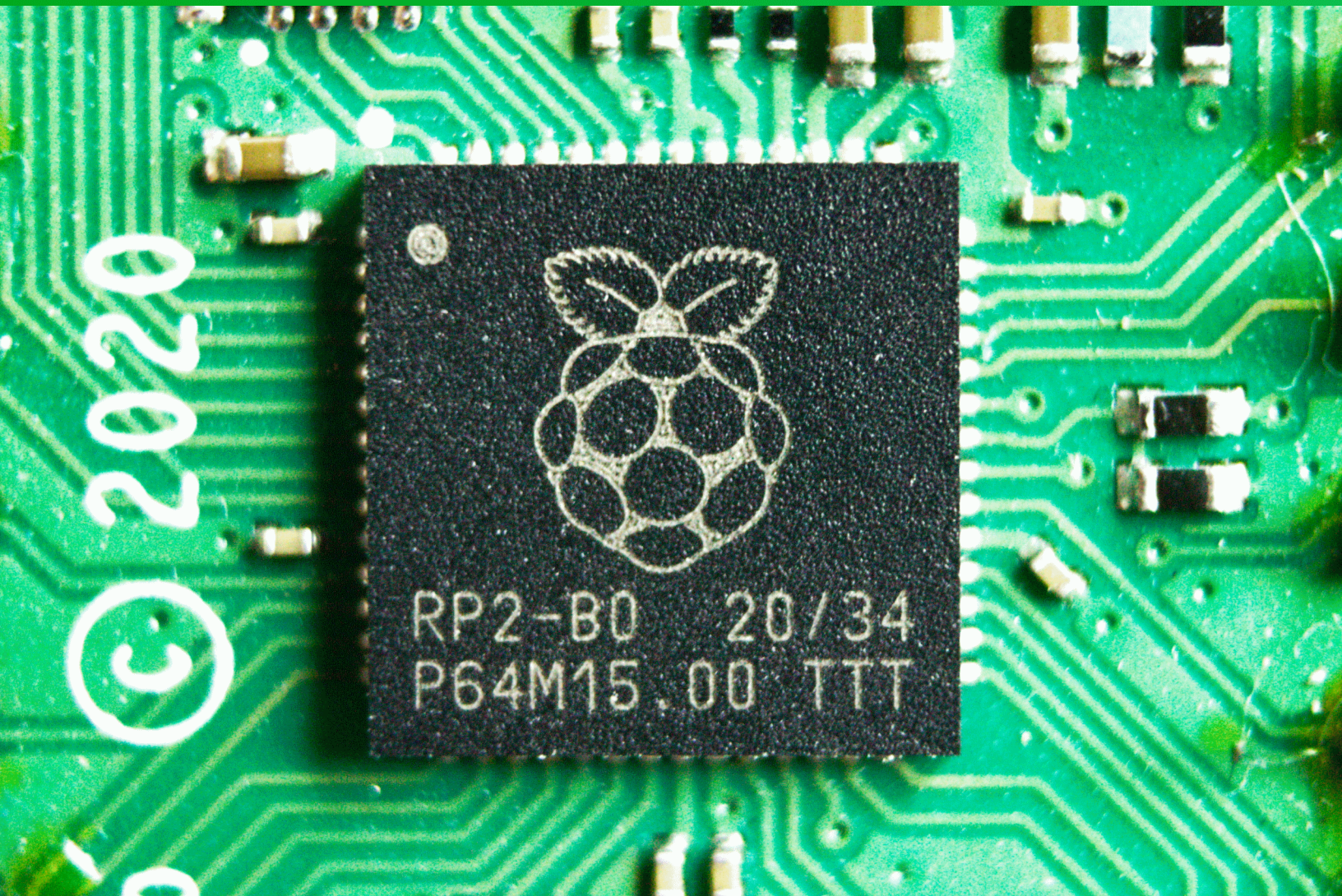


Knowing the RP2040

A Guide for Programmers



Daniel Quadros

Knowing the RP2040

A Guide for Programmers

Daniel Quadros

This book is for sale at <http://leanpub.com/rp2040>

This version was published on 2022-10-20



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2022 Daniel Quadros

Contents

Introduction	1
What this Book is About	1
The SDK functions	2
The Examples	2
Whom this Book is For	2
Acknowledgments	2
Updates	3
How to Send Feedback	3
The RP2040 Architecture	4
Processor Subsystem	4
Bus Fabric	5
Address Map	5
Clock Generation	6
Memory	6
PIO	6
Peripherals	7
IOs	7
Future RP Microcontrollers?	7
The Cortex-M0+ Processor Cores	8
Unprivileged and Privileged Execution	8
Debugger Support	8
Memory Protection Unit (MPU)	9
Instruction Set	9
SIO	13
Systick Timer	15
Selected SDK Functions	15
Example	17
Reset, Interrupts and Power Control	22
Reset	22
Interrupts	23
Power Control	30

CONTENTS

Memory, Addresses and DMA	37
Memory in the RP2040	37
Addresses	39
Direct Memory Access (DMA)	41
DMA Usage Examples	51
Clock Generation, Timer, Watchdog and RTC	63
Clock Generation	63
Timer	71
Watchdog	77
RTC	79
GPIO, Pad and PWM	84
GPIO Overview	84
Function Select	85
PADs	88
Digital Input and Output	91
GPIO Interrupts	103
PWM	109
The Programmable I/O (PIO)	127
The PIO State Machines	127
The FIFOs	127
Programmer's Model	128
PIO Configuration	128
Interrupt (IRQ) Flags	130
The Instructions	130
Flow Control	138
Coding, Compiling and Running PIO Programs	139
PIO Assembly Language	139
Selected SDK Functions	141
Examples	146
Communication Using I²C	154
I ² C Basics	154
I ² C in the RP2040	158
Selected SDK Functions	159
Examples	161
Asynchronous Serial Communication: the UARTs	167
Framing	167
FIFOs	168
Control Signals and Hardware Flow Control	168
Baud Rate Generation	169

CONTENTS

UART Status and Interrupts	170
Pins Options	171
Selected SDK Functions	171
Using the UART Registers	173
Example	174
Communication Using SPI	179
SPI Basics	179
SPI in the RP2040	180
Selected SDK Functions	182
Example	183
Analog Input: the ADC	189
Overview	189
Modes of Operation	189
Accuracy of the ADC	190
Temperature Sensor	190
Selected SDK Functions	191
Example	192
A Brief Introduction to the USB Controller	195
USB Basics	195
Hardware	198
Device Classes	199
TinyUSB	199
Using the USB	199
The HID Device Class	200
Example - Emulating a PC Keyboard	201
Example - Connecting a PC Keyboard to the Pi Pico	211
Example - Serial USB Adapter	219
Conclusion	228
Appendix A - CMake Files for RP2040 Programs	229
Appendix B - Using stdio	232
Selected pico_stdio Functions	233
Selected pico_stdio_uart Functions	233
Selected pico_stdio_usb Functions	234
The printf Function	234
Appendix C - Debugging Using the SWD Port	237
PicoProbe Connections	237
Software Installation	238
Debugging from the Command Line	239

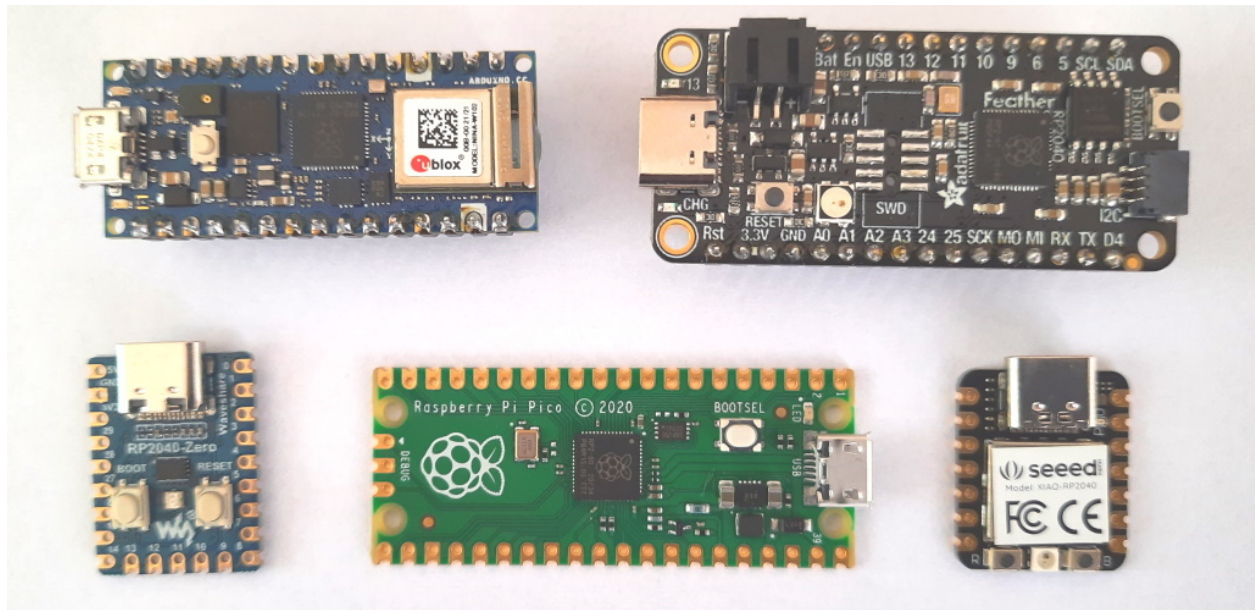
CONTENTS

Debugging from inside Visual Code	239
Appendix D - Accessing the RP2040 Registers	243
Registers Addresses and Basic Access	243
Special Write Operations	244
Using the SDK Functions for GPIO Output	246

Introduction

On January 21st 2021 the Raspberry Pi Foundation announced a microcontroller board, the Raspberry Pi Pico. At its heart there is a brand new microcontroller, the RP2040. A quick browsing on the specs will show that it's very powerful: two ARM M0+ running at 133MHz, 264kbytes of Ram, all the popular interfaces (UART, I2C, SPI, ADC) and a somewhat magical Programmable I/O (PIO) subsystem.

At the same time the Pi Pico was launched, a few other companies (close partners of the Raspberry Pi Foundation) announced their own boards with the RP2040. Later on, the RP2040 was made available to everyone and there is now more than a dozen boards on the market, with more sure to come.



A Few RP2040 Boards

What this Book is About

The Raspberry Pi Foundation provides some pretty good documentation, including a datasheet and an SDK user guide, so you may ask why I am writing this book.

The answer is that the official documentation is more about “what things are” than “why this is important” or “how do I use it”. I’ve tried to explain things in a logical and clear way so you can get a better knowing of what the RP2040 is capable of and how to use it.

This is not a “project book”, so code examples are short and focused on an specific feature. This is also not a “hardware book”, you will not found here much talk about designing a board around the RP2040 (but I will talk a little about hardware on some points).

The SDK functions

The C/C++ SDK includes many libraries. Most of the functions in these libraries provides a way to interact with the hardware, abstracting the low level registers in the RP2040.

I will not try to cover every function in the SDK. Instead I will focus on the functions I believe are the most useful for typical programs.

You can check the full list of SDK functions in the official documentation at <https://raspberrypi.github.io/pico-sdk-doxygen/>

The Examples

The examples where written in C, using the Pico SDK version 1.4.0. Like many, I am not particularly fond of the installation process for the SDK (specially on Windows) and the use of CMake may be a hurdle to those more accustomed to programming under the nice umbrella of an user-friendly IDE or know only about makefiles. But the Pico SDK is the official way to access the low level stuff we are going to see.

The examples were tested on a Pi Pico and should run on other boards, eventually with changes regarding the available pins.

All code from the examples can be download from <https://github.com/dquadros/KnowingRP2040>

Whom this Book is For

This is what I would call an **intermediate book**, going into a few advanced topics.

A assume the reader has a little experience with microcontrollers and some very basic knowledge of electronics.

Anyone who knows the basics of the C language should have no trouble understanding the examples.

Acknowledgments

Looking back, there are too many people that, one way or another, have helped me come to the point where a could write this book. This is where I mention and thank a few of them.

First, my mother and father who nurtured my curiosity and addicted me to reading.

There were many teachers that not only gave important lessons, but also encouraged me to learn more.

In my professional life I am grateful for all that believed I could deliver and those that helped me do so.

A special mention to the late Alberto Fabiano, who introduced me to the wild community of hackerspaces. And to Fabio Souza and Tiago Lima at Embarcados for all their work at spreading knowledge and their support to my technical writings.

Mauricio Aniche, my son-in-law, awarded professor and famous author, was a constant encourager when I was giving up to procrastination.

Of course this book would not exist if not for the patience of my wife Cecilia while I spent days in front of a PC and playing with all those “little boards”.

Updates

This is the first update after the book was “finished”, the changes include:

- A more precise explanation of the SIO registers
- Details on GPIO interrupts in Chapter 7, including a new example.
- The new Appendix D on how to access the RP2040 registers.
- Correction of typing errors and small improvements on explanations.

How to Send Feedback

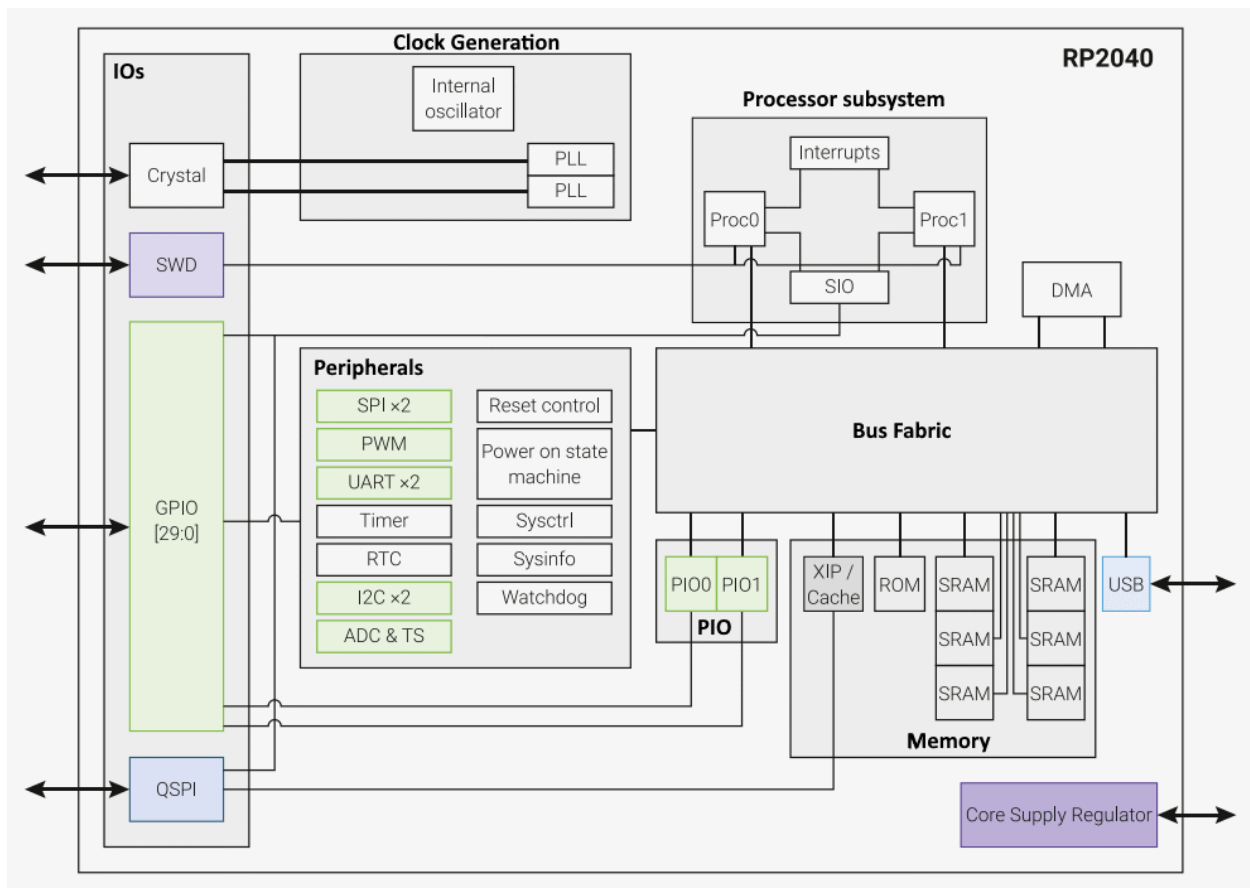
If you find an error, have a suggestion or comment, you can reach me by:

- Clicking in the “Email the Author” button in the book page in leanpub.com
- Sending an email to dqsoft.blogspot@gmail.com
- Sending me a message in Twitter ([@DQSoft](https://twitter.com/DQSoft))

The RP2040 Architecture

An overall view is usually a good starting point. When it comes to microcontrollers, an architectural diagram will show important things like how the memories are connect to the processor and what kind of peripherals are available.

The following picture is adapted from the RP2040 datasheet and shows the RP2040 architecture:



The RP2040 Architecture

In this chapter I will not delve deep into each part, I will just give a general idea of what they do and mention some important points. The next chapters will go into the details.

Processor Subsystem

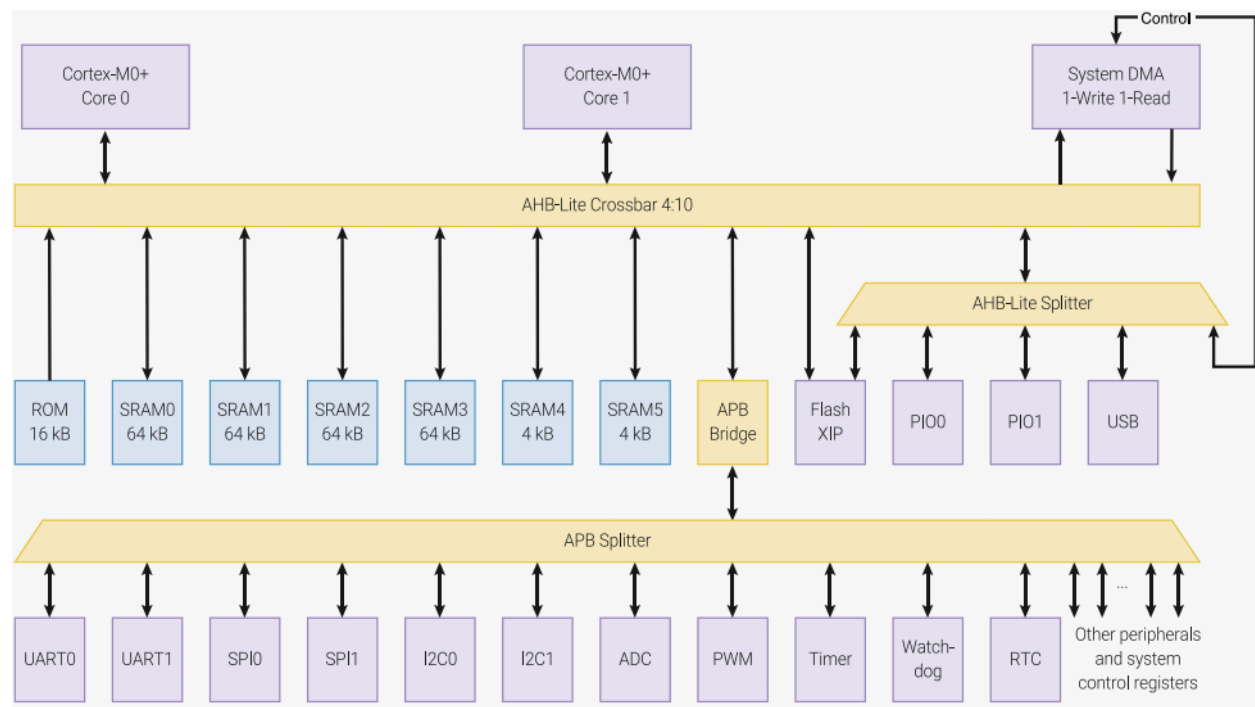
Let's start by the block called **Processor Subsystem**. Here we have the two **ARM Cortex M0+** cores and, connected to both, the **SIO** (Single-cycle IO block).

The ARM core is where the instructions in our software will be decoded and executed. As we will see later, the ARM M0+ is a very powerful RISC processor designed for embedded applications.

The SIO gives the cores low-latency, deterministic access to some peripherals (in the form of memory mapped registers), including the GPIO. In other words, the SIO makes it easier to write code that will run in both cores and use the same peripherals. It is designed for fast synchronization and communication between the two cores.

Bus Fabric

The **Bus Fabric** gives the processor cores access to memory and other peripherals. Like many ARM processors, there are two main buses, the **AHB** (called here *AHB-Lite Crossbar*) and the **APB**.



The RP2040 Bus Fabric

The AHB-Lite Crossbar routes addresses and data between 4 upstream ports (the two cores and DMA read and write) and 10 downstream ports. Up to four AHB bus transfers can take place each cycle.

The APB routes access to the more slower peripherals, an APB bridge connects the two buses.

Address Map

All resources on the RP2040 are mapped to memory addresses between 0x00000000 and 0xF0000000:

Address	Resource
0x00000000	ROM
0x10000000	XIP
0x20000000	SRAM
0x40000000	APB Peripherals
0x50000000	AHB-Lite Peripherals
0xD0000000	IOPORT Registers
0xE0000000	Cortex-M0+ Registers

Clock Generation

The RP2040 has a sophisticated clock generation subsystem. There are three basic clock sources:

- Ring oscillator: this internal oscillator needs no external components but has very vague frequency guarantees (typically 6MHz, expected between 4 and 8MHz, can be anywhere from 1.8 to 12 MHz, can change with voltage or temperature).
- Crystal oscillator: the hardware reference design (and all boards so far) have an external 12MHz crystal connected to this oscillator. The output of the crystal oscillator is fed to two PLLs that can generate higher frequencies for USB and system clock.
- External clocks: up to three clocks, with frequencies up to 50MHz.

These sources are connected to ten clock generators; each generator can be configured to select the source and the clock divisor. The output of these generators goes to the other subsystems.

Memory

In this subsystem we have the elements: RAM, ROM and XIP/Cache.

There are six RAM blocks (four with 64kB and two with 4kB). As we will see, this allows the two cores to access Ram with no contentions.

The 16kbytes ROM comes from factory with the startup firmware. Among other things, it will create an USB mass storage device used to write software to the Flash memory.

This Flash memory is not inside the RP2040. It is an external component, attached to GPIO pins controlled by the QSPI interface. Software can be run directly from the external Flash (eXecute In Place or XIP). To alleviate the delays of serial access to the external Flash, there is a 16kbytes cache.

PIO

The **Programmable Input Out** allows the RP2040 to efficiently implement many hardware protocols that need to be implemented by “bit-banging” (careful manual control of GPIOs pins) in other microcontrollers.

The RP2040 has two PIOs, each with 4 **state machines**. This state machines can interact with GPIOs by executing short programs. All the state machines run in parallel to the ARM processors. Data exchange between the state machines and the processors are done through hardware queues to alleviate timing requirements.

The result is a system where precise timing and constant signal monitoring can be achieved with minimum processor overhead, even when very short times are required.

Peripherals

The RP2040 has the following peripherals:

- 2 UARTs (Universal Asynchronous Receiver Transmitter)
- 2 SPI (Serial Peripheral Interface)
- 2 I²C (Inter-integrated Circuit)
- PWM (Pulse Width Modulation)
- ADC (Analog to Digital Converter)
- Timer
- RTC (Real Time Clock)

IOs

The IOs subsystem includes

- The crystal driver
- GPIOs - the general purpose I/Os and the pin drivers (called **Pad** in the documentation)
- QSPI - High speed SPI for connecting the Flash memory where firmware will reside
- SWD - Serial Wire Debug. This interface gives an external debug the ability to load software in Ram or Flash, control processor execution, access memory.

Future RP Microcontrollers?

While there is no official word (so far) about other RP microcontroller, the coding of the name gives some idea of what the Raspberry Foundation expects to change in future chips:

- Different number of cores. A single core could be an even cheaper option, more than 2 cores seem a little overkill, but who knows?
- A different type of core. The probable candidates would be the M3 (more computing power) and the M4 (DSP instructions and, optionally, floating point).
- More (or less) Ram. Less Ram would make the chip cheaper and/or open space to other features. More Ram may be interesting for a more powerful core type.
- Addition of nonvolatile memory, probably Flash. This would make boards simpler and also give faster access to nonvolatile code and data.

The Cortex-M0+ Processor Cores

The ARM Cortex-M is a group of processor cores developed by the ARM Holdings. They are based on the ARM 32 bit RISC processor and optimized for embedded applications.

There are many Cortex-M cores, the M0+ is an update of the Cortex-M0, the lowest cost option. The specs for the M0+ includes a few optional features (from the higher end M3 and M4) and the RP2040 implements most of them.

Key objectives of the Cortex-M0 are low cost, low energy consumption, high performance and deterministic interruption handling and instruction timing.

The key features of the M0+ cores in the RP2040 are:

- ARMv6-M architecture.
- Von Neumann architecture, where all memory can hold code and data (as opposed to the Harvard architecture, where there as separated memory for code and data).
- Thumb instructions (compact 16-bit encoding for a subset of the ARM instruction set) support.
- 32-bit single-cycle hardware multiplier.
- Low power sleep mode.
- 26 interrupts, plus a non-masked interrupt, with a relocatable vector table (the vector table holds the address for the interrupt handlers).
- Debugger support through the SerialWire debug interface.
- 8 region memory protection unit.
- 24-bit SysTick timer.

In this chapter we will take a look into some low-level stuff. Most applications will not have to deal with these.

Unprivileged and Privileged Execution

The RP2040 supports two modes of execution: Unprivileged and Privileged. After a reset code runs in the Privileged mode, but an OS can run code in the Unprivileged mode, restricting what it can do.

Debugger Support

A debugger, like gdb, can be connected using the two wire **SerialWire** interface (SWD). The following debug features are available:

- Upload a program into Ram or Flash.
- Access memory for inspection or change.
- Four hardware *breakpoints* (a breakpoint stops the application when execution reaches a specific address).
- Two *watchpoints* (a watchpoint stops the application when data is accessed at a specific address).
- Program Counter Sampling Register (PCSR) for non-intrusive code profiling. The PCSR registers the address of a “recently used” instruction, without stopping the program. By reading it periodically, a profiler can find out where a program is spending time.
- Single step and vector catch capabilities.

You can use another Pi Pico to connect a PC to the SerialWire interface, see Appendix C.

Memory Protection Unit (MPU)

The MPU protects the system address space by dividing the memory into regions and controlling access rights. It does not perform address translation. It is normally used by an OS to enforce privilege rules, separate process and manage memory attributes.

The MPU supports up to 8 regions (numbered 0 to 7), plus a default memory map. If the MPU is disabled, access is controlled by the default map. If the MPU is enabled, the default map can be used as a background region (numbered -1). The regions can overlap, with higher numbered regions having priority.

The size of a region must be a power of 2, from 2^8 to 2^{32} . Each region of size 2^n can be divided in up to 8 subregions of size 2^{n-3} . For example, we can define a region of size 512k (2^{19}) to cover all the 264k of SRAM and, inside it,

When an address is accessed, the MPU will check if its covered by one of the regions and, if yes, it will check permissions. If its not covered or the access does not pass the permission check, a fault is generated.

The permissions checked are the write, privileged and Execute Never (XN) memory attributes.

Instruction Set

The M0+ implements the ARMv6-M Thumb instruction set. This is a compact form of the original ARM instruction set (that is *not* supported). In Thumb a program is a stream of 16-bit halfwords, aligned at even addresses. Most instructions use only one halfword. A few 32-bit instruction require two halfwords.

Each core has a set of 32-bit registers:

- R0 to R12 are general use registers.

- R13 is the stack pointer. The CONTROL register selects between the Master Stack Pointer and the Process Stack Pointer
- R14 is the link register. It stores the return address for exceptions and subroutines. To nest subroutines the code must explicit save R14 in the stack.
- R15 is the program counter that points to the current instruction. Care must be taken when using R15 as a change in its content will cause a jump.
- PSR is the Program Status Register.
- PRIMASK controls the activation of exceptions.
- CONTROL selects the stack and the privilege level.

In most instructions, the THUMB encoding restricts register references to R0 to R7 (the LO registers, usually indicated by an ending 'S' in the instruction mnemonic). A few instructions can use R0 to R15 (the ANY registers).

The following table lists the instructions available. For details see the ARMv6-M Architecture Reference Manual at <https://developer.arm.com/documentation/ddi0419/latest>.

Operation	Assembler	Description
Move	MOVS Rd,#imed	Move 8-bit constant to Rd(0-7)
	MOVS Rd,Rm	Move Rm(0-7) to Rd (0-7)
	MOV Rd,Rm	Move Rm(0-15) to Rd(0-15)
Add	ADDS Rd,Rn,#imed	Add 3-bit constant to Rn(0-7) result in Rd(0-7)
	ADDS Rd,#imed	Add 8-bit constant to Rd(0-7) result in Rd
	ADDS Rd,Rn,Rm	Add Rn(0-7) to Rm(0-7) result in Rd(0-7)
	ADD Rd,Rn	Add Rn(0-15) to Rd(0-15) result in Rd
	ADD Rd,SP,#imed	Add constant to SP, result in Rd(0-7). Constant must be multiple of 4 in range 0-1020
	ADD SP,SP,#imed	Add constant to SP. Constant must be multiple of 4 in range 0-508
	ADD SP,Rm	Add Rm(0-15) to SP
	ADCS Rd,Rn	Add Rn(0-15) and carry to Rd(0-15) result in Rd
	ADR Rd,label	Load label address in Rn(0-15). Label address is codified as an offset from PC
Subtract	SUBS Rd,Rn,#immed	Subtract 3-bit constant from Rn(0-7) result in Rd(0-7)
	SUBS Rd,#immed	Subtract 8-bit constant from Rd(0-7) result in Rd
	SUBS Rd,Rn,Rm	Subtract Rm(0-7) from Rn(0-7) result in Rd(0-7)
	SBCS Rd,Rn,Rm	Subtract Rm(0-7) and carry from Rn(0-7) result in Rd(0-7)
	SUB SP,#imed	Subtract constant from SP. Constant must be multiple of 4 in range 0-508
	RSBS Rd,Rn,#0	Negate: Rd(0-7) = - Rn(0-7)
Multiply	MUL Rd,Rn	Multiply Rd(0-7) by Rn(0-7), result in Rd.

Operation	Assembler	Description
Compare	CMP Rn,Rm	update flags on Rn(0-15) - Rm(0-15)
	CMN Rn,Rm	update flags on Rn(0-7) + Rm(0-7)
	CMP Rn,#immed	update flags on Rn(0-7) - 8-bit constant
Logical	ANDS Rd,Rm	Rd(0-7) = Rd(0-7) AND Rm(0-7)
	EORS Rd,Rm	Rd(0-7) = Rd(0-7) XOR Rm(0-7)
	ORRS Rd,Rm	Rd(0-7) = Rd(0-7) OR Rm(0-7)
	BICS Rd,Rm	Rd(0-7) = Rd(0-7) AND NOT Rm(0-7)
	MVNS Rd,Rm	Rd(0-7) = NOT Rm(0-7)
	TST Rn,Rm	update flags on Rn(0-7) AND Rm(0-7)
	LSLS Rd,Rm,#shift	logical shift left Rm(0-7) by shift(0-31) bits, result in Rd(0-7)
Shift	LSLS Rd,Rs	logical shift left Rd(0-7) by Rs(0-7) bits
	LSRS Rd,Rm,#shift	logical shift right Rm(0-7) by shift(1-32) bits, result in Rd(0-7)
	LSRS Rd,Rs	logical shift right Rd(0-7) by Rs(0-7) bits
	ASRS Rd,Rm,#shift	arithmetic shift right Rm(0-7) by shift(1-32) bits, result in Rd(0-7)
	ASRS Rd,Rs	arithmetic shift right Rd(0-7) by Rs(0-7) bits
	RORS Rd,Rs	rotate right Rd(0-7) by Rs(0-7) bits
	LDR Rd,[Rn+#immed]	load Rd(0-7) with the word at address Rn(0-7)+immed (0-124, multiple of 4)
Load	LDRH Rd,[Rn+#immed]	load Rd(0-7) with the halfword at address Rn(0-7)+immed (0-62, multiple of 2)
	LDRB Rd,[Rn,#immed]	load Rd(0-7) with the word at address Rn(0-7)+immed (0-31)
	LDR Rd,[Rn,Rm]	load Rd(0-7) with the word at address Rn(0-7)+Rm(0-7)
	LDRH Rd,[Rn,Rm]	load Rd(0-7) with the halfword at address Rn(0-7)+Rm(0-7)
	LDRSH Rd,[Rn,Rm]	load Rd(0-7) with the signed halfword at address Rn(0-7)+Rm(0-7)
	LDRB Rd,[Rn,Rm]	load Rd(0-7) with the byte at address Rn(0-7)+Rm(0-7)
	LDRSB Rd,[Rn,Rm]	load Rd(0-7) with the signed byte at address Rn(0-7)+Rm(0-7)
	LDR Rd,label	load Rd(0-7) with the word at label. Label is coded as an offset (0-1020, multiple of 4) from PC
	LDR Rd,[SP,#immed]	load Rd(0-7) with the word at address SP+immed(0-1020, multiple of 4)
	LDM Rn!,loreglist	loads multiple registers(0-7) from words starting at the address in Rn(0-7). Rn must not be in loreglist and it is updated with the next address
	LDM Rn,loreglist	loads multiple registers(0-7) from words starting at the address in Rn(0-7). Rn must be in loreglist and it receives the loaded value

Operation	Assembler	Description
Store	STR Rd,[Rn,#immed]	store Rd(0-7) in the word at address Rn(0-7)+immed (0-124, multiple of 4)
	STRH Rd,[Rn,#immed]	store low halfword of Rd(0-7) at address Rn(0-7)+immed (0-62, multiple of 2)
	STRB Rd,[Rn,#immed]	store low byte of Rd(0-7) at address Rn(0-7)+immed (0-31)
	STR Rd,[Rn,Rm]	store Rd(0-7) in the word at address Rn(0-7)+Rm(0-7)
	STRH Rd,[Rn,Rm]	store low halfword of Rd(0-7) at address Rn(0-7)+Rm(0-7)
	STRB Rd,[Rn,Rm]	store low byte of Rd(0-7) at address Rn(0-7)+Rm(0-7)
	STR Rd,[SP,#immed]	store Rd(0-7) at address SP+immed(0-1020, multiple of 4)
	STM R!,loreglist	store multiple registers(0-7) starting at the address in Rn(0-7). Rn is updated with the next address
Push	PUSH loreglist	push registers(0-7) in the stack
	PUSH loreglist,LR	push registers(0-7) and LR (R14) in the stack
POP	POP loreglist	pop registers(0-7) from the stack
	POP loreglist,PC	pop registers(0-7) from the stack and return
Branch	Bcc label	conditional branch, label must be within -252 to +258 bytes of current instruction
	B label	unconditional branch, label must be within 2 kbytes of current instruction
	BL label	save next instruction address in LR and branch to label. This is a 32 bit instruction, label can be within 4Mbytes of current instruction
	BX Rm	branch to Rm AND 0xFFFFFFFF
	BLX Rm	save next instruction address in LR and branch to Rm AND 0xFFFFFFFF
		branch to Rm AND 0xFFFFFFFF
Extend	SXTH Rd,Rm	extend signed low halfword in Rm(0-7), result in Rd(0-7)
	SXTB Rd,Rm	extend signed low byte in Rm(0-7), result in Rd(0-7)
	UXTH Rd,Rm	extend unsigned low halfword in Rm(0-7), result in Rd(0-7)
	UXTB Rd,Rm	extend unsigned low byte in Rm(0-7), result in Rd(0-7)
Reverse	REV Rd,Rm	reverse bytes in Rm(0-7), result in Rd(0-7)
	REV16 Rd,Rm	reverse bytes in each halfword in Rm(0-7), result in Rd(0-7)
	REVSH Rd,Rm	reverse halfwords in Rm(0-7), result in Rd(0-7)
State	SVC #immed	Supervisor call. Generates an exception, typically used for requesting an OS service, selected by immed (0-255).

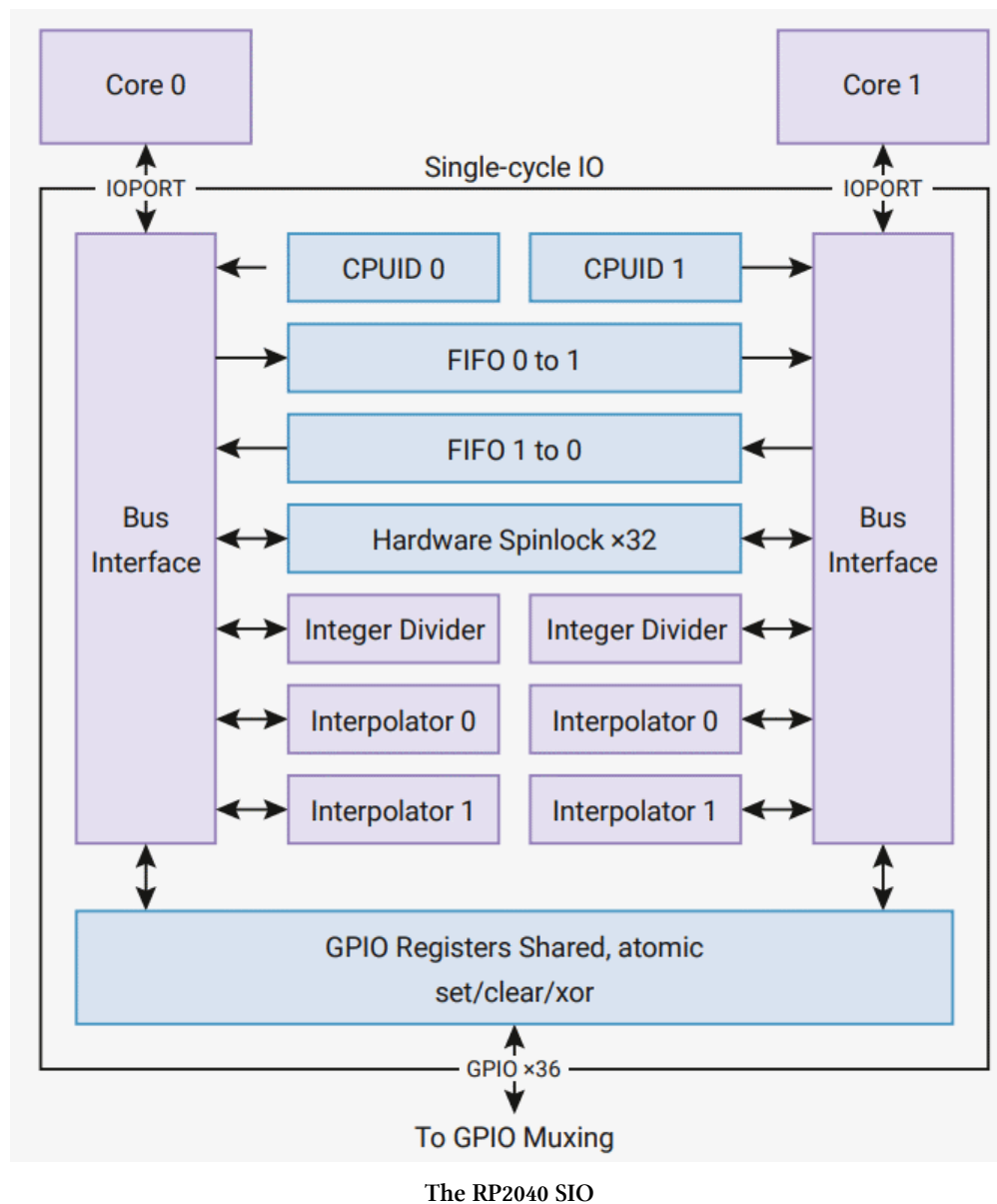
Operation	Assembler	Description
	CPSID i	Disable interrupts
	CPSIE i	Enable interrupts
	MRS Rd,specreg	Read special register specreg, result in Rd(0-7). This is a 32 bit instruction.
	MSR specreg,Rn	Write Rd(0-7) to a special register. This is a 32 bit instruction.
	BKPT #immed	Breakpoint, causes a debug halt. immed(0-255) can be used by the debugger to identify the breakpoint.
Hint	SEV	Send event
	WFE	Wait for event
	WFI	Wait for interrupt
	YIELD	Indicate task is waiting and could be swapped
Barriers	NOP	No operation
	ISB	Instruction Synchronization Barrier. This is a 32 bit instruction.
	DMB	Data Memory Barrier. This is a 32 bit instruction.
	DSB	Data Synchronization Barrier. This is a 32 bit instruction.

Notes:

- “XXXX Rd,Rn” can also be written as “XXXX Rd,Rd,Rn”
- The ranges of the constants come from shifting them right and limiting to a few bits in the instruction encoding
- Some instructions have multiple encoding to support LO and ANY variants
- A word is 32 bit, a halfword is 16 bit
- LDM and STM encode the affected registers in a bit field, so the order in memory is the numeric order of the registers not the order in the lorelist
- conditions for branch are EQ, NE, CS/HS, CC/LO, MI, PL/VS, VC, HI, LS, GE, LT, GT and LE
- special registers are APSR, IAPSR, EAPSR, XPSR, IPSR, EPSR, IEPSR, MSP, PSP, PRIMASK and CONTROL
- Hint instructions can be used by an OS to help implement multithreading
- Barriers are used to force synchronization events by the processor with respect to retiring load or store instructions. A memory barrier is used to guarantee completion of preceding load or store instructions, flushing of any prefetched instructions prior to the event, or both

SIO

The SIO (single-cycle IO block) contains memory-mapped hardware that need to be accessed quickly and concurrency-safely by the two cores.



The RP2040 registers can be accessed atomically through SET, CLR and XOR addresses. As long as you use these addresses to update the registers, there is no risk of concurrency problems between the two core (or interrupts). If you use separate instructions to read a SIO register, generate the new value and write back it, you can have problems if the other core or an interrupt changes the same register. Appendix D talks more about accessing the SIO registers.

There are two features of special interest when writing multicore software using the C/C++ SDK: the Hardware Spinlocks and the Inter-processor FIFOs.

Hardware Spinlocks

The hardware spinlocks are 32 one bits flags, each mapped to a different memory address. After writing any value to a spinlock the next read will return a non zero value. All other reads will return zero.

Spinlocks are used to manage exclusive access to resources that are locked for very short periods of time. Before an access the software will read the spinlock until it gets a non zero value. After the access the software does a write on the spinlock to release the resource.

Inter-processor FIFOs

The SIO has two First In First Out queues, each with 8 words of 32 bits. One FIFO can only be written by core 0 and read by core 1 and the other goes in the opposite direction.

The two processor can check if data is available in the FIFO by reading a status register or be alerted by an interrupt.

Systick Timer

The *Systick* is a 24-bit counter decremented by the `timer_tick` (1 MHz). A register defines the value loaded when the count reaches zero, an interrupt can be generated when this happens.

Selected SDK Functions

pico_multicore

The `pico_multicore` library contains functions for running code on core 1 and support for the FIFOs. After a reset, the runtime of the SDK will run `main()` in core0 and put core1 to sleep.

```
void multicore_reset_core1 (void)
```

Resets core1.

```
void multicore_launch_core1(void (*entry)(void))
```

Wake up core1 and runs `entry(void)` on it. Core 1 must be reset before calling this function. The interrupt vector will be the same as core 0. Uses the default core 1 stack. There are other functions that give more control of the interrupt vector and stack.

```
void multicore_fifo_drain(void)
```

Discards all data in the read FIFO.

```
uint32_t multicore_fifo_pop_blocking (void)
```

Wait indefinitely for data available in the read FIFO and read it.

```
bool multicore_fifo_pop_timeout_us (uint64_t timeout_us, uint32_t *out)
```

Wait for data available in the read FIFO or a timeout. Returns true if data was read and copied to out, false if timed-out.

```
void multicore_fifo_push_blocking (uint32_t data)
```

Waits indefinitely for space in the write queue and write data to it.

```
bool multicore_fifo_push_timeout_us (uint32_t data, uint64_t timeout_us)
```

Wait for space on the write FIFO or timeout. Returns true if wrote data to the FIFO, false if timeout.

pico_sync

The pico_sync library implements four types of synchronization primitives:

- **critical_section**: for use in normal code and interrupt handlers, prevents execution to be interrupted by the other core or higher priority interrupts. As interrupts are blocked, the section should be very short. Implemented using a spinlock.
- **lock_core**: base synchronization primitive for mutex and semaphores.
- **mutex**: for use in normal (non-interrupt) code, typically used to protect data structures. To access the data structure you first request ownership of the mutex (and wait if the other core has it). After using the data structure, you release the ownership. There are two type of mutex: normal and recursive, the difference been that the owner of a normal mutex will block if it tries to get ownership again (dead-lock).
- **sem**: used to restrict access to resources. A semaphore has a count of available resources, when you acquire it this count is decremented and when you release it the count is incremented. In typical use you will block execution if you try to acquire a resource that is not available and resume execution when the resource is made available. Acquiring should be done only in normal code, releasing can be done in normal and interrupt code.

```
void critical_section_init (critical_section_t *crit_sec)
```

Initialize a critical section. Must be called before the other functions.

```
static void critical_section_enter_blocking (critical_section_t *crit_sec)
```

Checks the spinlock until it is free, then grab it. Use to wait for permission to enter the critical section.

```
static void critical_section_exit (critical_section_t *crit_sec)
```

Release the spinlock, indicating that the critical section was exited.

```
void mutex_init (mutex_t *mtx)
```

```
void recursive_mutex_init (recursive_mutex_t *mtx)
```

Initialize the mutex. Must be called before the other functions.

```
void mutex_enter_blocking (mutex_t *mtx)
```

```
void recursive_mutex_enter_blocking (recursive_mutex_t *mtx)
```

Blocks until the caller can take ownership of the mutex.

```
bool mutex_try_enter (mutex_t *mtx, uint32_t *owner_out)
```

```
bool recursive_mutex_try_enter (recursive_mutex_t *mtx, uint32_t *owner_out)
```

Checks if the mutex is free. If it is, takes ownership and returns true. If its taken, returns false.

```
bool mutex_enter_timeout_ms (mutex_t *mtx, uint32_t timeout_ms)
```

```
bool recursive_mutex_enter_timeout_ms (recursive_mutex_t *mtx, uint32_t timeout_ms)
```

```
bool mutex_enter_timeout_us (mutex_t *mtx, uint32_t timeout_us)
```

```
bool recursive_mutex_enter_timeout_us (recursive_mutex_t *mtx, uint32_t timeout_us)
```

```
bool mutex_enter_block_until (mutex_t *mtx, absolute_time_t until)
```

```
bool recursive_mutex_enter_block_until (recursive_mutex_t *mtx, absolute_time_t until)
```

This are variations of the enter_blocking with timeout. Return true if the caller got ownership of the mutex, false if timeout occurred.

```
void mutex_exit (mutex_t *mtx)
```

```
void recursive_mutex_exit (recursive_mutex_t *mtx)
```

Release the mutex.

```
void sem_init (semaphore_t *sem, int16_t initial_permits, int16_t max_permits)
```

Initialize a semaphore. initial_permits is the number of resources available at the time of creation, max_permits is the maximum number of resources possible.

```
void sem_reset (semaphore_t *sem, int16_t permits)
```

Resets a semaphore, permits is the new current number of resources available.

```
int sem_available (semaphore_t *sem)
```

Returns the number of resources available. Not very useful if resources can also be consumed by the other core.

```
bool sem_release (semaphore_t *sem)
```

Releases a resource.

```
void sem_acquire_blocking (semaphore_t *sem)
```

Consumes a resource, waiting if none available.

```
bool sem_acquire_timeout_ms (semaphore_t *sem, uint32_t timeout_ms)
```

```
bool sem_acquire_timeout_us (semaphore_t *sem, uint32_t timeout_us)
```

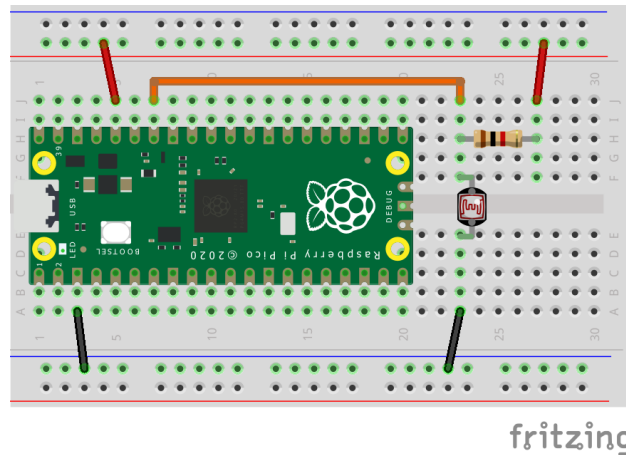
```
bool sem_acquire_block_until (semaphore_t *sem, absolute_time_t until)
```

Tries to consume a resource, waiting for one available or timeout. Returns true if a resource was consumed or false if timed out.

Example

In this example both cores will use the ADC. Core 1 will read the internal temperature and pass the result to Core 0. Core 0 will measure GPIO28 and print an average of the readings in stdio. A mutex is used to control the use of the ADC and the FIFO is used to pass the temperature reading from core 1 to core 0.

To try this program you can use an LDR connected to GPIO28:



Circuit for dual core example

Here is the code:

Dual Core Example

```

1  /**
2   * @file dualcore.c
3   * @author Daniel Quadros
4   * @brief Example of using the two ARM cores in the RP2040
5   *        A mutex is used to control usage of the ADC
6   *        An interprocessor FIFO is used to pass data between the cores
7   * @version 0.1
8   * @date 2022-06-03
9   *
10  * @copyright Copyright (c) 2022, Daniel Quadros
11  *
12  */
13
14  #include <stdio.h>
15  #include <string.h>
16  #include <stdlib.h>
17
18  #include "pico/stdlib.h"
19  #include "pico/multicore.h"
20  #include "pico/sync.h"
21  #include "hardware/gpio.h"
22  #include "hardware/adc.h"
23
24  // Where the LDR is connected
25  #define GPIO_LDR      28

```

```
26 #define ADC_INPUT_LDR    2
27
28 // Internal temperature sensor
29 #define ADC_INPUT_TEMPSENSOR 4
30
31 // Mutex for ADC
32 mutex_t adc_mutex;
33
34 // Factor to convert ADC reading to voltage
35 // Assumes 12-bit, ADC_VREF = 3.3V
36 const float conversionFactor = 3.3f / (1 << 12);
37
38 // This routine will run in core 1
39 void readRpTemp() {
40     while(1) {
41         // Get access to the ADC
42         mutex_enter_blocking(&adc_mutex);
43
44         // Select ADC input and read temperature sensor voltage
45         adc_select_input(ADC_INPUT_TEMPSENSOR);
46         adc_read(); // throw away first reading after changing input
47         uint16_t adc = adc_read();
48
49         // Release the ADC
50         mutex_exit(&adc_mutex);
51
52         // Convert reading to temperature in units of 0.1 C
53         float tempC = 27.0f - (adc*conversionFactor - 0.706f) / 0.001721f;
54         int32_t tempDC = (int32_t) ((tempC * 10.0f) + 0.5f);
55
56         // Pass the value to the other core
57         multicore_fifo_push_blocking(tempDC);
58     }
59 }
60
61 // Main Program
62 int main() {
63     // Init stdio
64     stdio_init_all();
65     printf("\nDual Core Example\n");
66
67     // Init ADC
68     adc_init();
```

```
69     adc_set_temp_sensor_enabled(true);
70     mutex_init (&adc_mutex);
71
72     // Make sure GPIO is high-impedance, no pullups etc
73     adc_gpio_init(GPIO_LDR);
74
75     // Start other core
76     multicore_launch_core1(readRpTemp);
77
78     // Main loop
79     const int MAX_COUNT = 10000;
80     int count = 0;
81     float tempSum = 0.0f;
82     float ldrSum = 0.0f;
83     while (1) {
84         // Get access to the ADC
85         mutex_enter_blocking(&adc_mutex);
86
87         // Select ADC input and read LDR voltage
88         adc_select_input(ADC_INPUT_LDR);
89         adc_read(); // throw away first reading after changing input
90         uint16_t adc = adc_read();
91
92         // Release the ADC
93         mutex_exit(&adc_mutex);
94
95         // Convert reading to voltage and accumulate
96         ldrSum += adc * conversionFactor;
97
98         // Get a temperature reading and accumulate
99         tempSum += multicore_fifo_pop_blocking()*0.1f;
100
101         // Print out the averages after MAX_COUNT readings
102         if (++count == MAX_COUNT) {
103             printf("LDR voltage: %.2f V Temperature: %.2f\n", ldrSum/MAX_COUNT, tem\
104 pSum/MAX_COUNT);
105             count = 0;
106             tempSum = 0.0f;
107             ldrSum = 0.0f;
108         }
109     }
110 }
```

For compiling the code you will need the CMakeLists.txt:

CMakeList.txt for Dual Core Example

```
1 cmake_minimum_required(VERSION 3.13)
2
3 include(pico_sdk_import.cmake)
4
5 project(dualcore_project)
6
7 pico_sdk_init()
8
9 add_executable(dualcore
10     dualcore.c
11 )
12
13
14 target_link_libraries(dualcore PRIVATE
15     pico_stdlib
16     pico_multicore
17     pico_sync
18     hardware_adc
19     hardware_gpio
20 )
21
22 pico_enable_stdio_usb(dualcore 1)
23 pico_enable_stdio_uart(dualcore 0)
24
25 pico_add_extra_outputs(dualcore)
```

Reset, Interrupts and Power Control

Reset

The full reset of the RP2040 (*chip-level reset*) puts it in the starting state. It can be caused by:

- initial power on.
- a *brown-out event* (power supply dropping below a certain voltage). After a reset the nominal brown out threshold is 0.86V, this threshold can be changed and the brownout detector can be disabled under software control.
- the RUN pin being put at LOW level.
- through the SWD bus. There is a *Rescue DP* (debug port) available over the SWD bus that is only intended for use in the specific case where the chip has locked up. The Rescue DP is reset by the other means, but not when itself causes the reset.

The RP2040 has a register that informs the cause of the most recent reset.

The reset controller also allows the software to reset all peripherals (except for a few that are critical).

Selected SDK Functions

The `hardware_resets` library has the following functions:

```
static void reset_block (uint32_t bits)
```

Reset the blocks selected by `bits`. The blocks stay reseted until an `unreset_` function is called.

The table below shows the bit for each block:

Block	Bit	Block	Bit
USB	24	ADC0	0
UART 1	23	Bus Control	1
UART 0	22	DMA	2
Timer	21	I2C 0	3
TB Manager	20	I2C 1	4
SysInfo	19	IO Bank 0	5
System Config	18	IO Bank 1	6
SPI 1	17	JTAG	7
SPI 0	16	Pads - Bank 0	8
RTC	15	Pads - QSPI	9
PWM	14	PIO 0	10
PLL USB	13	PIO 1	11
PLL System	12		


```
static void unreset_block (uint32_t bits)
```

Removes the reset from the blocks selected by bits. It may take some time for the blocks to complete the initialization.

```
static void unreset_block_wait (uint32_t bits)
```

Removes the reset from the blocks selected by bits and waits for the blocks to complete the initialization.

Interrupts

Interrupt signals go to the *Nested Vectored Interrupt* Controller (NVIC). The NVIC decides when to dispatch the interrupt to a handler routine, based on priorities that can be set by software. The handling of an interrupt can itself be interrupted by a higher-priority interrupt (so we can have *nested* interrupts).

The addresses of the routines that handle the interrupts are stored in a table in memory, the *vector table*.

A companion to the NVIC is the *Wakeup Interrupt Controller* (WIC). When the RP2040 is in DORMANT state, the WIC is responsible for the identification of interrupts and waking up the processor to attend to them (more details in the Power Control section).

Interrupts in the NVIC are numbered from 0 to 31, but the RP2040 only uses the lower 26. The table below lists the interrupts, the names used in the source column are defined in `intctrl.h`

IRQ	Source	IRQ	Source
0	TIMER_IRQ_0	13	IO_IRQ_BANK0
1	TIMER_IRQ_1	14	IO_IRQ_QSPI
2	TIMER_IRQ_2	15	SIO_IRQ_PROC0
3	TIMER_IRQ_3	16	SIO_IRQ_PROC1
4	PWM_IRQ_WRAP	17	CLOCKS_IRQ
5	USBCTRL_IRQ	18	SPI0_IRQ
6	XIP_IRQ	19	SPI1_IRQ
7	PIO0_IRQ_0	20	UART0_IRQ
8	PIO0_IRQ_1	21	UART1_IRQ
9	PIO1_IRQ_0	22	ADC0_IRQ_FIFO
10	PIO1_IRQ_1	23	I2C0_IRQ
11	DMA_IRQ_0	24	I2C1_IRQ
12	DMA_IRQ_1	25	RTC_IRQ

Each processor core has an NVIC, the same interrupts are available to both cores. **An interrupt should be enabled in just one core.**

Selected SDK Functions

The `hardware_irq` library has the functions dealing with interrupts. This functions affect only the NVIC of the core that calls them.

An interrupt handler routine must be a void function with no parameters.

There are three ways to attach a handler routine to an interrupt:

- Defining statically the handler explicit by declaring a void function with no parameters and a name like `isr_dma_0` (in this case for `dma_irq_0`). This is not recommended as it offers no advantage and may cause link errors if another module also defines the same function.
- Use the `irq_set_exclusive_handler()` function to attach the handler at runtime. This should be use for interrupts that will have only one handler.
- Use the `irq_set_shared_handler()` function to attach the handler at runtime. This is useful when an interrupt is shared by multiple sources (like `IO_IRQ_BANK0` for GPIO interrupts) and you want to have multiple handlers. This incurs in a small time penalty, as a library function will receive the interrupt and call the registered handlers.

```
void irq_set_priority (uint num, uint8_t hardware_priority)
```

Sets the priority (0 to 255) of a hardware interrupt. Lower values mean higher priority. By default all interrupts have the priority set to `PICO_DEFAULT_IRQ_PRIORITY` (0x80).

```
void irq_set_enabled (uint num, bool enabled)
```

Enables (`enabled = true`) or disables (`enabled = false`) an interrupt.

```
bool irq_is_enabled (uint num)
```

Returns true if the interrupt is enabled.

```
void irq_set_mask_enabled (uint32_t mask, bool enabled)
```

Enables (`enabled = true`) or disables (`enabled = false`) the interrupts indicated by `mask` (each bit correspond to an interrupt, bit 0 is `TIMER_IRQ_0` and so on).

```
void irq_set_exclusive_handler (uint num, irq_handler_t handler)
```

Assign handler to handle the interrupt corresponding to `num`. Will trigger an assert if there is already a handle assigned.

```
void irq_add_shared_handler (uint num, irq_handler_t handler, uint8_t order_priority)
```

Add a handler to an interrupt. Handlers will be called in descending order of `order_priority` (the order is undefined for equal priorities).

Notice that all the handlers will be called. Each one should check (and clear) an specific cause for the interrupt (see the example).

```
void irq_remove_handler (uint num, irq_handler_t handler)
```

Remove a shared handler.

```
static void irq_clear (uint int_num)
```

Clears a pending interrupt.

```
void irq_set_pending (uint num)
```

Set an interrupt as pending (will call the handler if the interrupt is enabled). This normally not used, as interrupts are generated by the hardware.

Examples

In chapter 9 (Asynchronous Serial Communication: the UARTs) there is a simple example of using interrupts with the UART.

The following example shows how to use shared handlers and PIO (Programmable I/O) interrupts. More details on the PIO can be found in the corresponding chapter. For now you need to know that a PIO has four *State Machines*, each capable of running special programs simultaneously to the each other and the ARM cores.

Notice: this example will not run with SDK versions prior to 1.4.0 as there was a bug in irq_add_shared_handler.

This example uses a PIO program that generates periodic interrupts. The period is set by writing into the PIO Tx FIFO the number of cycles to wait between interrupts. The C program runs this program on two State Machines, with different periods.

Interrupts will be generated and handled in this example as follows:

- Interrupts are generated by the **IRQ** instruction in the PIO code. Each PIO has 7 interrupt flags, we are using “0 rel” in the IRQ instruction, meaning “flag 0 plus the state machine number”, so we get different flag when running the same program in different state machines.
- In the state machine configuration, we are enabling the interrupt flag as a reason to generate the `PIO0_IRQ_0` interrupt.
- We set two shared handlers for `PIO0_IRQ_0`, each handler will check one state machine for interrupts.
- If a handler detects that a state machine generated the interrupt, it clears it (as this is not automatic).

CMakeLists.txt for PIO Interrupt Example

```
1 cmake_minimum_required(VERSION 3.13)
2
3 include(pico_sdk_import.cmake)
4
5 project(pioint_project)
6
7 pico_sdk_init()
8
9 add_executable(pioint
10     pioint.c
11 )
12
13 pico_generate_pio_header(pioint ${CMAKE_CURRENT_LIST_DIR}/pioint.pio)
14
15 target_link_libraries(pioint PRIVATE
16     pico_stdlib
17     pico_sync
18     hardware_pio
19     hardware_irq
20 )
21
22 pico_enable_stdio_usb(pioint 1)
23 pico_enable_stdio_uart(pioint 0)
24
25 pico_add_extra_outputs(pioint)
```

PIO Code

```
1 ;
2 ; Periodic interrupts - Example for 'Knowing the RP2040' book
3 ; Copyright (c) 2022, Daniel Quadros
4 ;
5
6 .program pioint
7
8     pull                // get delay
9     mov y, osr          // save delay in Y
10 .wrap_target
11 loop1:
12     mov x, y            // load delay in counter
13 loop2:
```

```

14     jmp x-- loop2    // loop delay cycles
15     irq 0 rel        // interrupt
16 .wrap
17
18
19 % c-sdk {
20 // Helper function to set a state machine to run our PIO program
21 static inline void pioint_program_init(PIO pio, uint sm, uint offset,
22     float freq) {
23
24     // Get an initialized config structure
25     pio_sm_config c = pioint_program_get_default_config(offset);
26
27     // Configure the clock
28     float div = clock_get_hz(clk_sys) / freq;
29     sm_config_set_clkdiv(&c, div);
30
31     // Enable our interrupt at IRQ0
32     pio_set_irq0_source_enabled(pio, pis_interrupt0 + sm, true);
33
34     // Clear IRQ flag before starting
35     pio_interrupt_clear(pio, sm);
36
37     // Load our configuration, and jump to the start of the program
38     pio_sm_init(pio, sm, offset, &c);
39
40     // Set the state machine running
41     pio_sm_set_enabled(pio, sm, true);
42 }
43 %}

```

C Code

```

1 /**
2  * @file pioint.c
3  * @author Daniel Quadros
4  * @brief Example of using PIO interrupts
5  * @version 0.1
6  * @date 2022-08-17
7  *
8  * @copyright Copyright (c) 2022, Daniel Quadros
9  *
10 */

```

```
11
12 #include "stdio.h"
13 #include "pico/stdlib.h"
14 #include "pico/sync.h"
15 #include "hardware/irq.h"
16 #include "hardware/pio.h"
17 #include "hardware/clocks.h"
18
19 // Our PIO program:
20 #include "pioint.pio.h"
21
22 // Flag to signal interrupts received
23 volatile int intRx = 0;
24
25 // critical section for accessing the flag
26 critical_section_t cs_intRx;
27
28 // PIO and State Machines
29 PIO pio = pio0;
30 int sm1, sm2;
31
32 // Rx interrupt handler for sm1
33 void on_sm1_int() {
34     if (pio_interrupt_get(pio, sm1)) {
35         pio_interrupt_clear(pio, sm1);
36         intRx |= 1;
37     }
38 }
39
40 // Rx interrupt handler for sm2
41 void on_sm2_int() {
42     if (pio_interrupt_get(pio, sm2)) {
43         pio_interrupt_clear(pio, sm2);
44         intRx |= 2;
45     }
46 }
47
48
49 // Main routine
50 int main() {
51     // Start stdio and wait for USB connection
52     stdio_init_all();
53     while (!stdio_usb_connected()) {
```

```
54     sleep_ms(100);
55 }
56 printf ("PIO Interrupt demo\n");
57
58 // Init the critical section
59 critical_section_init (&cs_intRx);
60
61 // Find a location (offset) in the instruction memory where there is
62 // enough space for our program and load it there
63 uint offset = pio_add_program(pio, &pioint_program);
64
65 // Find a free state machine on our chosen PIO
66 // Configure it to run our program, and start it, using the
67 // helper function we included in our .pio file.
68 sm1 = pio_claim_unused_sm(pio, true);
69 ppoint_program_init(pio, sm1, offset, 200000.0f);
70 printf ("SM1 = %d\n", sm1);
71
72 // Find another free state machine on our chosen PIO
73 // Configure it to run our program, and start it, using the
74 // helper function we included in our .pio file.
75 sm2 = pio_claim_unused_sm(pio, true);
76 ppoint_program_init(pio, sm2, offset, 200000.0f);
77 printf ("SM2 = %d\n", sm2);
78
79 // Set up the interrupt handlers
80 irq_add_shared_handler(PIO0_IRQ_0, on_sm1_int, 1);
81 irq_add_shared_handler(PIO0_IRQ_0, on_sm2_int, 2);
82 irq_set_enabled(PIO0_IRQ_0, true);
83
84 // The state machines are now running.
85 // Set the delays and start interrupts
86 pio_sm_put_blocking (pio, sm1, 400000);
87 pio_sm_put_blocking (pio, sm2, 700000);
88
89 // Loop testing for interrupts
90 int flags;
91 while (true) {
92     sleep_ms(1);
93
94     // Copy and clear interrupt flag
95     critical_section_enter_blocking(&cs_intRx);
96     flags = intRx;
```

```
97         intRx &= ~3;
98         critical_section_exit(&cs_intRx);
99
100        // Print message if interrupt received
101        if (flags & 1) {
102            printf("<< INT SM1 >>\n");
103        }
104        if (flags & 2) {
105            printf("<< INT SM2 >>\n");
106        }
107    }
108 }
```

Power Control

Power consumption is a main concern in microcontroller projects, specially when battery operation is required. To obtain high performance **and** low power consumption, not only low consumption components are required but also software control over the power usage.

This allows us to implement systems where there are occasional short bursts of high power consumption while keeping power consumption low most of the time.

The RP2040 provides a few features for reducing power consumption:

- Some peripherals can be powered down, e.g. the temperature sensor in the ADC.
- Clock can be stopped for individual peripherals and functional blocks. This can be done automatically based on processor sleep state.
- The system clock source and (for some sources) the clock frequency can be changed without stopping the processor.
- Memories can be put into a state-retaining power down state.

Top-level Clock Gates

The top-level clock gates control the clocks for the endpoints of each clock signal. For example, the `clk_peri` feeds the SPI and UART peripherals, each one has an independent clock gate.

The state of a peripheral is maintained if its clock is temporarily removed by a clock gate. No reset or reinitialization is required when the clock is re-enabled.

There are two registers that control the clock gates, one determines the clocks that remain enabled when the RP2040 is placed in the SLEEP mode and the other the clocks enabled when the RP2040 is awake.

Sleep States

The RP2040 has two sleep states: **SLEEP** and **DORMANT** (or **DEEPSLEEP**).

The **SLEEP** state is entered when both processors are asleep and there is no outstanding system DMA transfers. **SLEEP** state is exited when a processor is awoken by an interrupt. **SLEEP** mode is useful when we can stop processing while waiting for a interrupt from one (or more) of the internal peripherals. We set up the clock gates so that clock is provided only for the peripherals that can wake the system.

The **DORMANT** state is more aggressive: all clock and all oscillators are disabled. **DORMANT** mode can only be exit by a GPIO event or an RTC interrupt. To use the RTC it must have same kind of external clock source (notice that **XOSC** is also disabled by **DORMANT**).

Selected SDK Functions

Functions to put the RP2040 into sleep and dormant states are not part of the C/C++ SDK, but can be find in `pico-extras` and are considered “work in progress”. The corresponding examples are in `pico-playground`. Both can be downloaded from <https://github.com/raspberrypi>.

```
void sleep_run_from_dormant_source(dormant_source_t dormant_source)
```

This function changes all clock sources to `dormant_source` to prepare for sleep or dormant state.

Values for `dormant_source` are `DORMANT_SOURCE_XOSC` and `DORMANT_SOURCE_ROSC`. As a shortcut you can call `sleep_run_from_xosc()` or `sleep_run_from_rosc()`.

```
void sleep_goto_sleep_until(datetime_t *t, rtc_callback_t callback)
```

Puts the RP2040 in sleep state until the specified time. One of the `sleep_run_from_*` routines must be called before. `callback` will be called after the RP2040 wakes.

```
void sleep_goto_dormant_until_pin(uint gpio_pin, bool edge, bool high)
```

Puts the RP2040 in dormant state until the specified gpio interrupt. One of the `sleep_run_from_*` routines must be called before.

`gpio_pin` is the pin number. `edge` and `level` select the event that will wake up the RP2040:

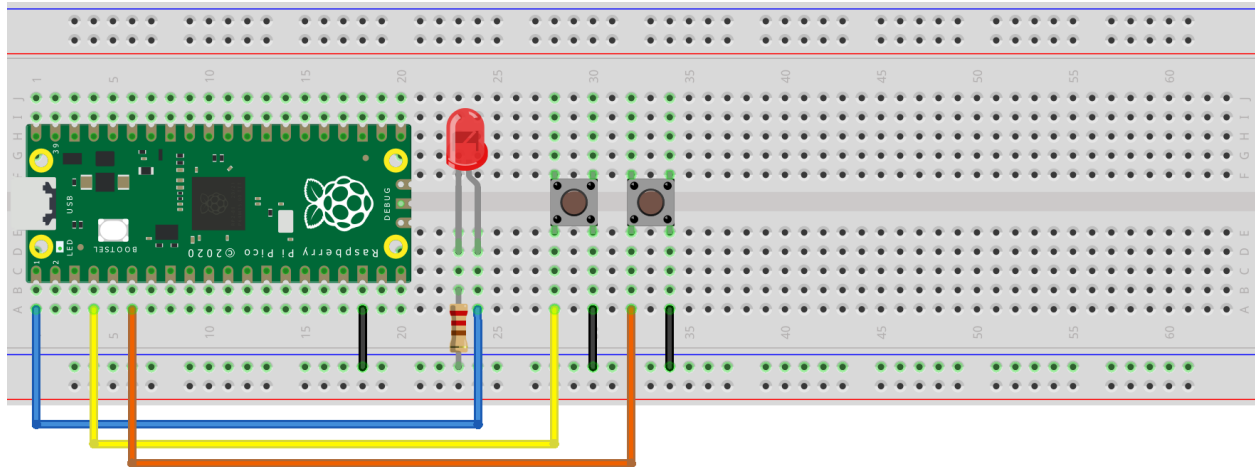
edge	level	event
false	false	pin is LOW
false	true	pin is HIGH
true	false	pin changes from HIGH to LOW
true	true	pin changes form LOW to HIGH

The functions `sleep_goto_dormant_until_edge_high(uint gpio_pin)` and `sleep_goto_dormant_until_level_high(uint gpio_pin)` are defined as `sleep_goto_dormant_until_pin(gpio_pin, true, true)` and `sleep_goto_dormant_until_pin(gpio_pin, false, false)`

Example

In this example we use two buttons and an LED to play with the power control. The LED will blink while the RP2040 is awake. Pressing and releasing the first button will put the RP2040 to sleep for 5 seconds. Pressing and releasing the second button will put the RP2040 in dormant mode until the second button is pressed and released again.

The diagram bellow shows how to connect the LED and switches.



fritzing

Circuit for sleep example

CMakeLists.txt for Sleep Example

```
1 cmake_minimum_required(VERSION 3.13)
2
3 include(pico_sdk_import.cmake)
4 include(pico_extras_import.cmake)
5
6 project(sleep_project)
7
8 pico_sdk_init()
9
10 add_executable(sleep
11     sleep.c
12 )
13
14 target_link_libraries(sleep
15     pico_stdlib
16     pico_time
17     hardware_sleep
18 )
```

```

19
20 pico_add_extra_outputs(sleep)

```

Code for Sleep Example

```

1  /**
2   * @file sleep.c
3   * @author Daniel Quadros
4   * @brief Example of using the SLEEP and DORMANT states
5   * @version 0.1
6   * @date 2022-08-18
7   *
8   * @copyright Copyright (c) 2022, Daniel Quadros
9   *
10  * This examples is an adaptation of the hello_sleep and
11  * hello_dormant examples in pico_playground.
12  *
13  */
14
15  #include "pico/stdlib.h"
16  #include "pico/sleep.h"
17  #include "pico/time.h"
18
19  #include "hardware/gpio.h"
20  #include "hardware/rtc.h"
21
22  // GPIO connections
23  #define LED 0
24  #define BTN1 2
25  #define BTN2 4
26
27  // For button detection and debounce
28  typedef enum { NOT_PRESSED, DBC_PRESS, PRESSED, DBC_RELEASE } BTN_STATE;
29
30  // Aux routine to get milliseconds since boot
31  static inline uint32_t board_millis(void) {
32      return to_ms_since_boot(get_absolute_time());
33  }
34
35  // This routine will be called when the RTC wakes the RP2040
36  static void sleep_callback(void) {
37  }
38

```

```
39 // Puts the RP2040 to sleep for 5 seconds
40 static void rtc_sleep(void) {
41     // Arbitrary start on 31 March 2021 18:00:00
42     datetime_t t = {
43         .year  = 2021,
44         .month = 05,
45         .day   = 31,
46         .dotw  = 3, // 0 is Sunday
47         .hour  = 18,
48         .min   = 00,
49         .sec   = 00
50     };
51
52     // Start the RTC to our arbitraty start
53     rtc_init();
54     rtc_set_datetime(&t);
55
56     // Sleep 5 seconds
57     t.sec = 5;
58     sleep_goto_sleep_until(&t, &sleep_callback);
59 }
60
61
62 // Main routine
63 int main() {
64
65     // We will run from XOSC
66     sleep_run_from_xosc();
67
68     // Init the GPIO pins
69     gpio_init(LED);
70     gpio_set_dir(LED, true);
71     gpio_init(BTN1);
72     gpio_set_dir(BTN1, false);
73     gpio_pull_up(BTN1);
74     gpio_init(BTN2);
75     gpio_set_dir(BTN2, false);
76     gpio_pull_up(BTN2);
77
78     // Main loop
79     uint32_t ledTime = board_millis();
80     bool ledValue = false;
81     BTN_STATE btn1State = NOT_PRESSED;
```

```
82     BTN_STATE btn2State = NOT_PRESSED;
83     uint32_t btnTime;
84     while (true) {
85         // Blink LED every 300 ms (if awake)
86         if (board_millis() > ledTime) {
87             ledValue = !ledValue;
88             gpio_put(LED, ledValue);
89             ledTime = board_millis() + 300;
90         }
91
92         // Check button 1
93         if (btn2State == NOT_PRESSED) {
94             switch (btn1State) {
95                 case NOT_PRESSED:
96                     if (!gpio_get(BTN1)) {
97                         btn1State = DBC_PRESS;
98                         btnTime = board_millis() + 100;
99                     }
100                     break;
101                 case DBC_PRESS:
102                     if (board_millis() > btnTime) {
103                         btn1State = PRESSED;
104                     }
105                     break;
106                 case PRESSED:
107                     if (gpio_get(BTN1)) {
108                         btn1State = DBC_RELEASE;
109                         btnTime = board_millis() + 100;
110                     }
111                     break;
112                 case DBC_RELEASE:
113                     if (board_millis() > btnTime) {
114                         btn1State = NOT_PRESSED;
115                         // Button was pressed and released
116                         // Sleep
117                         gpio_put(LED, false);
118                         rtc_sleep();
119                     }
120                     break;
121             }
122         }
123
124         // Check button 2
```

```
125     if (btn1State == NOT_PRESSED) {
126         switch (btn2State) {
127             case NOT_PRESSED:
128                 if (!gpio_get(BTN2)) {
129                     btn2State = DBC_PRESS;
130                     btnTime = board_millis() + 100;
131                 }
132                 break;
133             case DBC_PRESS:
134                 if (board_millis() > btnTime) {
135                     btn2State = PRESSED;
136                 }
137                 break;
138             case PRESSED:
139                 if (gpio_get(BTN2)) {
140                     btn2State = DBC_RELEASE;
141                     btnTime = board_millis() + 100;
142                 }
143                 break;
144             case DBC_RELEASE:
145                 if (board_millis() > btnTime) {
146                     btn2State = NOT_PRESSED;
147                     // Button was pressed and released
148                     // Put in dormant mode until button 1 is released
149                     gpio_put(LED, false);
150                     sleep_goto_dormant_until_pin(BTN1, true, true);
151                     // Give some time for BTN1 release debounce
152                     busy_wait_ms(100);
153                 }
154                 break;
155         }
156     }
157
158 }
159
160 return 0;
161 }
```

Memory, Addresses and DMA

Memory in the RP2040

The RP2040 has access to three types of memory:

- **ROM:** 16k of internal read-only memory programmed at manufacturing.
- **SRAM:** 264k of internal static random access memory.
- **Flash:** up to 16M of external flash memory, accessed via a QSPI interface.

The ROM

The ROM is at address 0x00000000, it contains the code that is executed when the RP2040 is reseted and some utility functions.

Assuming no failures, there are three types of startup:

- Normal startup to code in Flash.
- Startup in *device mode* (Flash CSn forced low, for example by a BOOT button).
- Watchdog *boot-to-RAM* feature (see Watchdog chapter for more information).

When the RP2040 starts in device mode, it will appear as a USB Mass Storage device and a PICOBOOT Device (used for special functions). This mode is used to load a program in the Flash by copying a UF2 file into the Mass Storage Device.

The ROM also has utility functions for:

- Fast floating point calculations (as the M0+ cores do not have floating point hardware support).
- Fast bit counting and manipulation functions.
- Fast memory fill / copy functions

The SRAM

While the 264k SRAM is mapped in a single continuous memory address, physically it is divide in six banks. This division allows simultaneously access to SRAM by different masters (for example, DMA may access one bank while an ARM core is accessing another). Up to four 32-bit SRAM accesses can take place every system clock cycle (one per master).

There are four 64kB banks (organized as 16k of 32-bit words) and two 4kB banks (organized as 1K of 32-bit words). The first four banks can be accessed in two different ways in two different ranges of addresses:

- From 0x20000000 to 0x2003FFFF the SRAM is organized in a *stripped* way, mapping sequentially words from each bank:
 - 0x20000000 is word 0 from bank 0
 - 0x20000001 is word 0 from bank 1
 - 0x20000002 is word 0 from bank 2
 - 0x20000003 is word 0 from bank 3
 - 0x20000004 is word 1 from bank 0
 - and so on
- From 0x21000000 to 0x2103FFFF the SRAM is accessed as four 64kB regions, one for each bank:
 - 0x21000000 is word 0 from bank 0
 - 0x21000001 is word 1 from bank 0
 - 0x21010000 is word 0 from bank 1
 - and so on

The smaller banks can only be accessed in a non stripped way, at addresses 0x20040000 and 0x20041000).

In most cases you will not care about banks and use SRAM as a single 264kB region. Banking only matters if you are trying to squeeze the last drop of performance.

Last, there are two more dedicated RAM blocks that can be used in very special circumstances:

- The eXecute In Place (XIP) cache can be used as a 16kB memory block starting at 0x15000000, if you disable the caching. This only makes sense if you are running code from SRAM, as Flash access without the cache is very slow.
- The USB controller has a 4kB block of memory starting at 0x50100000. This can be used if are not using USB.

Flash Memory

The RP2040 has no internal Flash memory, so for most applications an external Flash chip (of up to 16MB) has to be used. The Flash is accessed via the **QSPI** interface using the execute-in-place (**XIP**) hardware.

The QSPI (Quad Serial Peripheral Interface), is a variant of SPI where four bits are transferred at each clock pulse. A full 32-bit word will need eight clock pulses.

The XIP hardware makes the serial interface transparent and includes a cache. Any read at a 16MB memory window starting at 0x10000000 will look up the data in the XIP cache, and generate a serial transfer if it is not there.

The XIP cache will normally allow program execution from Flash with minimum delays. On the other hand, if you have data in the Flash that you will use frequently (specially with DMA) you should copy it to SRAM (size permitting).

Addresses

Let's take a closer look at the RP2040 memory map. As we saw in Chapter 2, the basic map is:

Address	Resource
0x00000000	ROM
0x10000000	XIP
0x20000000	SRAM
0x40000000	APB Peripherals
0x50000000	AHB-Lite Peripherals
0xD0000000	IOPORT Registers
0xE0000000	Cortex-M0+ Registers

Here are a few details (more can be found in the RP2040 datasheet).

XIP

Address	Resource
0x10000000	XIP_BASE
0x11000000	XIP_NOALLOC_BASE
0x12000000	XIP_NOCACHE_BASE
0x13000000	XIP_NOCACHE_NOALLOC_BASE
0x14000000	XIP_CTRL_BASE
0x15000000	XIP_SRAM_BASE
0x18000000	XIP_SSI_BASE

The regions at 0x11000000 to 0x13000000 are mirrors to the 0x10000000 region but with different cache options.

The registers that control the XIP are at XIP_CTRL_BASE. The registers that control the SSI (that implements QSPI) are at XIP_SSI_BASE.

XIP_SRAM_BASE is the address where the XIP cache can be used as RAM, if cache is disabled.

SRAM

Address	Resource
0x20000000	SRAM_BASE (stripped banks 0 to 3)
0x20040000	SRAM4_BASE (bank 4)
0x20041000	SRAM5_BASE (bank 5)
0x21000000	SRAM0_BASE (bank 0)
0x21010000	SRAM1_BASE (bank 1)
0x21020000	SRAM2_BASE (bank 2)
0x21030000	SRAM3_BASE (bank 3)

APB Peripherals

Address	Resource
SYSINFO_BASE	0x40000000
SYSCFG_BASE	0x40004000
CLOCKS_BASE	0x40008000
RESETS_BASE	0x4000C000
PSM_BASE	0x40010000
IO_BANK0_BASE	0x40014000
IO_QSPI_BASE	0x40018000
PADS_BANK0_BASE	0x4001C000
PADS_QSPI_BASE	0x40020000
XOSC_BASE	0x40024000
PLL_SYS_BASE	0x40028000
PLL_USB_BASE	0x4002C000
BUSCTRL_BASE	0x40030000
UART0_BASE	0x40034000
UART1_BASE	0x40038000
SPI0_BASE	0x4003C000
SPI1_BASE	0x40040000
I2C0_BASE	0x40044000
I2C1_BASE	0x40048000
ADC_BASE	0x4004C000
PWM_BASE	0x40050000
TIMER_BASE	0x40054000
WATCHDOG_BASE	0x40058000
RTC_BASE	0x4005C000
ROSC_BASE	0x40060000
VREG_AND_CHIP_RESET_BASE	0x40064000
TBMAN_BASE	0x4006C000

AHB-Lite Peripherals

Address	Resource
DMA_BASE	0x50000000
USBCTRL_BASE	0x50100000
USBCTRL_DPRAM_BASE	0x50100000
USBCTRL_REGS_BASE	0x50110000
PIO0_BASE	0x50200000
PIO1_BASE	0x50300000
XIP_AUX_BASE	0x50400000

Direct Memory Access (DMA)

The RP2040 DMA controller is capable of moving data from and to memory (and memory mapped peripherals) independently from the processor. The processors can be executing other tasks (or sleeping to save energy) at the same time the DMA is transferring data.

Better still, the DMA has a higher throughput than a single processor, allowing up to one 32-bit read and one 32-bit write per clock cycle.

There are three uses for DMA:

- *Memory-to-peripheral*: when a peripheral signals that it is ready for more data, the DMA reads data from RAM or Flash and write it to the peripheral's transmit FIFO.
- *Peripheral-to-memory*: when a peripheral signals that it has data available, the DMA reads data from the peripheral's receive FIFO and write it into RAM.
- *Memory-to-memory*: DMA can also read data from one address in RAM and write to another address.

The peripherals that can work with DMA are PIO, SPI, UAT, PWM, I2C, ADC and XIP.

The control of a DMA transfer is done by a **DMA channel**, the RP2040 has 12 independent channels. Each channel has a set of registers to configure and monitor the transfers. Channels can be combined (*chained*) for more complex behaviors.

Channel Configuration

Each channel has four registers:

- The *CTRL* register configures:
 - The enable or disable of the channel
 - The size of the transfer (8, 16 or 32 bits).
 - The increment of the addresses after each read or write.
 - The peripheral data request (**DREQ**) that signals when the next read or write can occur.
 - An optional DMA channel that will be triggered when the current channel finishes the configured transfers.
- The *READ_ADDR* register contains the address for the next read.
- The *WRITE_ADDR* register contains the address for the next write.
- The *TRANS_COUNT* register is used to control the number of transfers to be done. A read will get the remaining transfers in the current sequence. *A write to TRANS_COUNT will set the count for the next transfer sequence.*

The addresses will normally point to Ram, Flash or a peripheral's FIFO. They must be aligned to the transfer size.

The control of the address increments has four components:

- INCR_READ (1 bit): 1 if the read address will be incremented.
- INCR_WRITE (1 bit): 1 if the write address will be incremented.
- RING_SEL (1 bit): selects the address that will be affected by RING_SIZE, 1 if write e 0 if read.
- RING_SIZE (4 bits): defines address wrap. If 0, address wrap is not performed. If > 0, the increment will only affect the lower n bits of the address.

It is recommended to set the addresses and count at start of each sequences of transfers. *If not, the sequence will use the addresses at the end of the previous sequence and the count will start with the last written value.*

Configuring and Starting a Channel

Each DMA channel register can be accessed through four addresses. To understand why, we need to talk about how the channel can be configured and how a channel is started.

The obvious way to configure a channel is for the software to write the configuration directly to the registers. Another way is to store the configuration in memory and use another DMA channel to load the configuration in the registers. The DMA channel configuration in memory is called a *control block*. This second option is interesting when we have many transfers to do (creating a control block list) and we use *chaining* to automatically load the next control block when a sequence of transfers finishes.

There are three ways to start (*trigger*) a channel:

- Writing a non-zero value to one of its registers in a specific address (so it is treated as a *trigger register*).
- Setting another channel to chain to it when the sequence is finished.
- By writing in the MULTI_CHAN_TRIGGER register, this can start multiple channels at once.

A trigger will not start a channel if it is disabled or already running.

So, back to the registers addresses. Here are the offsets that can be used to access the registers:

Base offset	+0x0	+0x04	+0x08	+0x0C (Trigger)
0x00 (Alias 0)	READ_ADDR	WRITE_ADDR	TRANS_COUNT	CTRL_TRIG
0x10 (Alias 1)	CTRL	READ_ADDR	WRITE_ADDR	TRANS_COUNT_TRIG
0x20 (Alias 2)	CTRL	TRANS_COUNT	READ_ADDR	WRITE_ADDR_TRIG
0x30 (Alias 3)	CTRL	WRITE_ADDR	TRANS_COUNT	READ_ADDR_TRIG

One of the four addresses is used to make a write in the register start the channel. By having the registers in different order as address are incremented, we can make control blocks more compact by loading only some of the registers.

A pair of examples may make this clearer:

Single Transfer

In this example, we just want to configure and start a sequence of DMA transfers.

We can do this by writing `READ_ADDR`, `WRITE_ADDR`, `TRANS_COUNT` and `CTRL_TRIG`, in this order, using the address in “Alias 0”. The sequence is started when `CTRL_TRIG` is triggered.

Transferring a Series of Fixed Size Buffers to a Peripheral

Now suppose we have some buffers, at non consecutive addresses, with data we need to transmit using a peripheral. The number of bytes is the same in all buffers.

The transfer of each buffer is done by a DMA with the buffer address as the read address and the peripheral transmit FIFO address as the write address (we will disable the increment of the write address).

To fully automate the transfers, we create a series of control blocks in memory, one for each buffer, and set a first DMA channel to transfer this control blocks to the registers of a second DMA channel that will transfer the data in the buffer.

We could put the four parameters (read address, write address, transfer count and control) in each control block, but only the read address will change. We can optimize it by programing the fixed parameters (using non trigger address), putting only the read address in the control blocks and setting the write address of the first channel to the `READ_ADDR_TRIG` of the second channel.

There are two more details on the use of a list of control blocks:

- To mark the end of the list, we put one more control block with zero in the trigger register (a *null trigger*).
- Normally an interrupt is generated at the end of each sequence of transfers. A bit in the `CTRL` register changes this behavior to only generate interrupts for null triggers.

Data Request (DREQ) and Pacing Timers

A critical aspect for DMA performance and reliability is *when* a transfer is done. While in some cases the answer is “as fast as possible” (for example, memory to memory DMA), most peripherals produce and consume data at their own pace and sometimes we want to execute transfers in a periodic way.

Transfers are paced by Transfer Requests (TREQ). The options for the TREQ of a DMA channel are:

- a request from a device (DREQ)
- a request from one of four timers
- permanent (“as fast as possible”)

The RP2040 peripherals have short FIFOs that can accommodate small variances in timing, but the DMA needs a signal from them to avoid under or overflow of the FIFOs. This signal is the Data Request (DREQ). There are 40 options for DREQ for a channel:

DREQ	Name	DREQ	Name
0	DREQ_PIO0_TX0	20	DREQ_UART0_TX
1	DREQ_PIO0_TX1	21	DREQ_UART0_RX
2	DREQ_PIO0_TX2	22	DREQ_UART1_TX
3	DREQ_PIO0_TX3	23	DREQ_UART1_RX
4	DREQ_PIO0_RX0	24	DREQ_PWM_WRAP0
5	DREQ_PIO0_RX1	25	DREQ_PWM_WRAP1
6	DREQ_PIO0_RX2	26	DREQ_PWM_WRAP2
7	DREQ_PIO0_RX3	27	DREQ_PWM_WRAP3
8	DREQ_PIO1_TX0	28	DREQ_PWM_WRAP4
9	DREQ_PIO1_TX1	29	DREQ_PWM_WRAP5
10	DREQ_PIO1_TX2	30	DREQ_PWM_WRAP6
11	DREQ_PIO1_TX3	31	DREQ_PWM_WRAP8
12	DREQ_PIO1_RX0	32	DREQ_I2C0_TX
13	DREQ_PIO1_RX1	33	DREQ_I2C0_RX
14	DREQ_PIO1_RX2	34	DREQ_I2C1_TX
15	DREQ_PIO1_RX3	35	DREQ_I2C1_RX
16	DREQ_SPI0_TX	36	DREQ_ADC
17	DREQ_SPI0_RX	37	DREQ_XIP_STREAM
18	DREQ_SPI1_TX	38	DREQ_XIP_SSITX
19	DREQ_SPI1_RX	39	DREQ_XIP_SSIRX

The logic in the DMA (*credit-based DREQ*) keeps a count of requests not yet issued, so that it can make full use of the FIFO. For this to work a DREQ cannot be used in more than one channel and a peripheral FIFO must not be accessed while being used by DMA.

The four pacing timers for DMA have fractional (X/Y) divisor. A request is made at the rate ((X/Y) * sys_clk). X and Y are 16-bit numbers and X/Y must be less or equal one (limiting the rate to sys_clk).

Interrupts

A channel can generate an interrupt when:

- It finishes the configured number of transfers, OR
- It receives a *null trigger* (a zero is written in a trigger register)

DMA interrupts can be enabled or disabled on a per-channel basis. There are two system IRQ to where enabled DMA interrupts can be routed. This allows us to give different priority to some channels and/or divide interrupts between the two cores.

CRC Calculation

The DMA can observe (*sniff*) the data transferred on one channel and update a CRC (Cyclic Redundancy Code) accumulator. The CRCs supported are CRC-32, CRC-16 CCITT, parity and 32-bit checksum.

The accumulator register can be written to initialize the calculation. When reading the result, there are options for inverting and reverting the bits and to swap the bytes. This options affect only the reading of the register, not the calculation.

Selected SDK Functions

The DMA functions are in the `hardware_dma` library.

Each library (`hardware_XXX`), for peripherals that support DMA and have multiple instances, has a function named `dma_get_XXX_dreq()` that returns the number of the DREQ for that peripheral.

Channel Allocation

The library maintains a simple (but multicore safe) control of channel and timer usage:

```
int dma_claim_unused_channel (bool required)
```

This function will return the number of an unused DMA channel. This is the preferred way to select a DMA channel, as it avoids conflicts that may result if you use a fixed number.

If `required` is false, the function will return -1 if all channels are in use (*claimed*). If `required` is true and there is no free channel, the function will panic (send an error message to `stdio` and halt).

```
void dma_channel_claim (uint channel)
```

Marks DMA channel number `channel` as in use. Will cause a panic if the channel is already claimed.

```
void dma_claim_mask (uint32_t channel_mask)
```

Marks multiple DMA channels as in use. Will cause a panic if any the channels is already claimed.

Each bit in `channel_mask` corresponds to a channel: bit 0 to channel 0, bit 1 to channel 1 and so on.

```
void dma_channel_unclaim (uint channel)
```

Indicates that DMA channel number `channel` is no longer in use.

```
void dma_unclaim_mask (uint32_t channel_mask)
```

Indicates that multiple DMA channels are no longer in use.

Each bit in `channel_mask` corresponds to a channel: bit 0 to channel 0, bit 1 to channel 1 and so on.

```
bool dma_channel_is_claimed (uint channel)
```

Returns true if DMA channel number `channel` is claimed.

```
int dma_claim_unused_timer (bool required)
```

This function will return the number of an unused DMA timer. This is the preferred way to select a DMA timer, as it avoids conflicts that may result if you use a fixed number.

If `required` is false, the function will return -1 if all timers are in use. If `required` is true and there is no free timer, the function will panic.

```
void dma_timer_claim (uint timer)
```

Marks DMA timer number `timer` as in use. Will cause a panic if the timer is already claimed.

```
void dma_timer_unclaim (uint timer)
```

Indicates that DMA timer number `timer` is no longer in use.

```
bool dma_timer_is_claimed (uint timer)
```

Returns true if DMA timer number `timer` is claimed.

Channel Configuration Manipulation

The `dma_channel_config` object holds the configuration of a DMA channel. This object should be manipulated using the following functions.

```
static dma_channel_config dma_channel_get_default_config (uint channel)
```

This function returns a `dma_channel_config` object with a default configuration for a channel. This is the standard way to start creating an specific configuration.

```
static dma_channel_config dma_get_channel_config (uint channel)
```

Gets a `dma_channel_config` object filled with the current configuration of a channel.

```
static void channel_config_set_read_increment (dma_channel_config *c, bool incr)
```

Sets the `read_increment` property in a channel configuration. If `incr` is true, read address will be increment after each transfer.

```
static void channel_config_set_write_increment (dma_channel_config *c, bool incr)
```

Sets the `write_increment` property in a channel configuration. If `incr` is true, read address will be increment after each transfer.


```
static void channel_config_set_dreq (dma_channel_config *c, uint dreq)
```

Sets the DREQ number in a channel configuration.

```
static void channel_config_set_chain_to (dma_channel_config *c, uint chain_to)
```

Sets in the configuration the DMA channel that will be triggered when the current channel finishes the configured transfers.

```
static void channel_config_set_transfer_data_size (dma_channel_config *c, enum dma_channel_transfer_size size)
```

Sets the transfer size in a channel configuration. Values available for size are DMA_SIZE_8, DMA_SIZE_16 and DMA_SIZE_32.

```
static void channel_config_set_ring (dma_channel_config *c, bool write, uint size_bits)
```

Sets the address wrapping properties in a channel configuration. If write is true, the wrapping is applied to the write address, otherwise it applies to the read address. size_bits is the number of bits that will be affected by the increment operation, 0 turns off wrapping.

```
static void channel_config_set_bswap (dma_channel_config *c, bool bswap)
```

Sets DMA byte swapping property in a channel configuration object.

No effect if transfer size is 8. For 16 bit transfer size, the two bytes of each halfword are swapped. For 32 bit transfer size, the four bytes of each word are swapped to reverse their order.

```
static void channel_config_set_irq_quiet (dma_channel_config *c, bool irq_quiet)
```

Sets the condition of interrupt generation in a channel configuration. If irq_quiet is false, an interrupt will be generated at the end of each programmed transfers. If irq_quiet is true, an interrupt will only be generated by a null transfer.

```
static void channel_config_set_high_priority (dma_channel_config *c, bool high_priority)
```

Changes the high_priority flag in a channel configuration. The scheduling of DMA channels first look at **all** the *high priority* channels and then a **single low priority**. In most cases this will not make a difference in throughput.

```
static void channel_config_set_enable (dma_channel_config *c, bool enable)
```

Enable or disable a channel in a channel configuration.

A disabled channel ignores triggers, stops new transfers and pause the current transfer (if any).

```
static void channel_config_set_sniff_enable (dma_channel_config *c, bool sniff_enable)
```

Changes the flag to engage the CRC calculation logic in a channel configuration.

To use the CRC calculation you need to enable it in the channel configuration and in the DMA (using dma_sniffer_enable()).

Channel Configuration

```
static void dma_channel_configure (uint channel, const dma_channel_config *config,
volatile void *write_addr, const volatile void *read_addr, uint transfer_count, bool
trigger)
```

This is the main function for setting up a DMA channel. The specified channel is configured with the properties in the `dma_channel_config` object, the read and write addresses and the transfer count are set.

If `trigger` is true, the channel is triggered and will start the transfers immediately.

```
static void dma_channel_set_config (uint channel, const dma_channel_config *config, bool
trigger)
```

Sets the configuration of a channel using the properties in a `dma_channel_config` object. If `trigger` is true, the channel is triggered and will start the transfers immediately.

As the configuration do not hold the addresses or the transfer count, they should be set prior to calling this function with `trigger` true.

```
static void dma_channel_set_read_addr (uint channel, const volatile void *read_addr,
bool trigger)
```

Sets the read address in a channel. If `trigger` is true, the channel is triggered and will start the transfers immediately.

This is useful if you do not need to change the other configuration parameters.

```
static void dma_channel_set_write_addr (uint channel, volatile void *write_addr, bool
trigger)
```

Sets the write address in a channel. If `trigger` is true, the channel is triggered and will start the transfers immediately.

This is useful if you do not need to change the other configuration parameters.

```
static void dma_channel_set_trans_count (uint channel, uint32_t trans_count, bool
trigger)
```

Sets the transfer count in a channel. If `trigger` is true, the channel is triggered and will start the transfers immediately.

This is useful if you do not need to change the other configuration parameters.

DMA Transfers Control

```
static void dma_channel_transfer_from_buffer_now (uint channel, const volatile void
*read_addr, uint32_t transfer_count)
```

Starts a DMA transfer from a buffer. You should have previously configured the channel and set the write address.

```
static void dma_channel_transfer_to_buffer_now (uint channel, volatile void *write_addr,
uint32_t transfer_count)
```

Starts a DMA transfer to a buffer. You should have previously configured the channel and set the write address.

```
static void dma_start_channel_mask (uint32_t chan_mask)
```

Starts transfers in multiple DMA channels at the same time.

Each bit in `chan_mask` corresponds to a channel: bit 0 to channel 0, bit 1 to channel 1 and so on.

```
static void dma_channel_start (uint channel)
```

Starts transfer in a DMA channel.

```
static void dma_channel_abort (uint channel)
```

Stops a DMA transfer. This function when return after the DMA has stopped. Notice that this may cause a transfer completion interrupt, even if the transfer was not completed (due to a bug in the RP2040).

DMA Interrupts

```
static void dma_channel_set_irq0_enabled (uint channel, bool enabled)
```

Enables (`enabled= true`) or disables (`enabled= false`) DMA channel interrupt on `DMA_IRQ_0` for channel number `channel`.

```
static void dma_set_irq0_channel_mask_enabled (uint32_t channel_mask, bool enabled)
```

Enables (`enabled= true`) or disables (`enabled= false`) DMA channel interrupt on `DMA_IRQ_0` for the channels specified in `channel_mask`.

```
static void dma_channel_set_irq1_enabled (uint channel, bool enabled)
```

Enables (`enabled= true`) or disables (`enabled= false`) DMA channel interrupt on `DMA_IRQ_1` for channel number `channel`.

```
static void dma_set_irq1_channel_mask_enabled (uint32_t channel_mask, bool enabled)
```

Enables (`enabled= true`) or disables (`enabled= false`) DMA channel interrupt on `DMA_IRQ_1` for the channels specified in `channel_mask`.

```
static void dma_irqn_set_channel_enabled (uint irq_index, uint channel, bool enabled)
```

Enables (`enabled= true`) or disables (`enabled= false`) DMA channel interrupt on `irq_index` (0 for `DMA_IRQ_0` or 1 for `DMA_IRQ_1`) for channel number `channel`.

```
static void dma_irqn_set_channel_mask_enabled (uint irq_index, uint32_t channel_mask, bool enabled)
```

Enables (`enabled= true`) or disables (`enabled= false`) DMA channel interrupt on `irq_index` (0 for `DMA_IRQ_0` or 1 for `DMA_IRQ_1`) for the channels specified in `channel_mask`.

```
static bool dma_channel_get_irq0_status (uint channel)
```

Checks if a particular channel is a cause of `DMA_IRQ_0`. A return of `true` means it is. As DMA has 12 channels and only 2 system interrupts, you may need to share an interrupt.

```
static bool dma_channel_get_irq1_status (uint channel)
```

Checks if a particular channel is a cause of DMA_IRQ_1. A return of true means it is. As DMA has 12 channels and only 2 system interrupts, you may need to share an interrupt.

```
static bool dma_irqn_get_channel_status (uint irq_index, uint channel)
```

Checks if a particular channel is a cause of interrupt irq_index (0 for DMA_IRQ_0 or 1 for DMA_IRQ_1). A return of true means it is. As DMA has 12 channels and only 2 system interrupts, you may need to share an interrupt.

```
static void dma_channel_acknowledge_irq0 (uint channel)
```

Acknowledges a channel IRQ, resetting it as the cause of DMA_IRQ_0.

```
static void dma_channel_acknowledge_irq1 (uint channel)
```

Acknowledges a channel IRQ, resetting it as the cause of DMA_IRQ_1.

```
static void dma_irqn_acknowledge_channel (uint irq_index, uint channel)
```

Acknowledges a channel IRQ, resetting it as the cause of interrupt irq_index (0 for DMA_IRQ_0 or 1 for DMA_IRQ_1).

```
static bool dma_channel_is_busy (uint channel)
```

Returns true if DMA channel number channel is currently busy.

```
static void dma_channel_wait_for_finish_blocking (uint channel)
```

Blocks execution until DMA channel number channel is not busy.

CRC Calculation (DMA Sniffer)

```
static void dma_sniffer_enable (uint channel, uint mode, bool force_channel_enable)
```

Enables DMA CRC calculation (*sniffing*) on channel channel.

mode selects the calculation

Mode	Calculation
0x00	CRC-32 (IEEE 802.3)
0x01	CRC-32 (IEEE 802.3) with bit reversed data
0x02	CRC-16 (CCITT)
0x03	CRC-16 (CCITT) with bit reversed data
0x0E	XOR / Parity (result is 1 if odd number of '1's
0x0F	32-bit checksum

To use the CRC calculation you also need to enable it in the channel. This can be do setting force_channel_enable true or using channel_config_set_sniff_enable() in the configuration. In the first case it is enable directly in the channel control register, in the second you have to apply the configuration eith dma_channel_configure().

```
static void dma_sniffer_set_byte_swap_enabled (bool swap)
```

Enables (swap = true) or disables (swap = false) the Sniffer byte swap function.

```
static void dma_sniffer_disable (void)
```

Disables the CRC calculation logic.

DMA Timers

```
static void dma_timer_set_fraction (uint timer, uint16_t numerator, uint16_t denominator)
```

Set the frequency divider for a DMA timer. The timer will run at `sys_clock * numerator / denominator`. denominator must be greater or equal numerator.

```
static uint dma_get_timer_dreq (uint timer_num)
```

Returns the DREQ for timer `timer_num`.

DMA Usage Examples

In these two examples we will see DMA been used to transfer data from and to peripherals. The details of the peripherals programming are in the corresponding chapters.

Collecting Data from the ADC using DMA

The main part of this example is using DMA to fill a buffer with the ADC readings of the internal temperature sensor. To spice things a bit, we will:

- Alternate between two buffers, so we can process data in one buffer while DMA is filling the other.
- Use the DMA interrupt to record when the transfer finished and to stop the ADC.
- Use the CRC calculation to sum the values for us.

This is an example where the read address (the peripheral FIFO) is fixed and the write address (in the buffer) is incremented. The transfer count is the size of the buffer (number of samples).

Collecting Data from the ADC using DMA

```

1  /**
2   * @file adcdma.c
3   * @author Daniel Quadros
4   * @brief Example of using DMA with the ADC in the RP2040 to read
5   *        the internal temperature sensor
6   * @version 0.1
7   * @date 2022-09-06
8   *
9   * @copyright Copyright (c) 2022, Daniel Quadros
10  *
11  */
12
13  #include <stdio.h>
14  #include <string.h>
15  #include <stdlib.h>
16
17  #include "pico/stdlib.h"
18  #include "hardware/adc.h"
19  #include "hardware/dma.h"
20
21  // Internal temperature sensor
22  #define ADC_INPUT_TEMPSENSOR 4
23
24  // DMA channel number
25  int dma_chan;
26
27  // Factor to convert ADC reading to voltage
28  // Assumes 12-bit, ADC_VREF = 3.3V
29  const float conversionFactor = 3.3f / (1 << 12);
30
31  // Buffers for ADC readings
32  #define N_SAMPLES 1000
33  int iBuf = 0; // buffer currently used by DMA
34  uint16_t buffer[2][N_SAMPLES];
35  uint32_t finishedXfer[2];
36
37
38  // This routine will run when DMA finishes filling a buffer
39  void dma_irq_handler() {
40    // Stop ADC
41    adc_run(false);
42    // Clear the interrupt request.

```

```

43     dma_hw->ints0 = 1u << dma_chan;
44     // Register when DMA finished
45     finishedXfer[iBuf] = to_ms_since_boot(get_absolute_time());
46 }
47
48 // Main Program
49 int main() {
50     // Init stdio
51     stdio_init_all();
52     printf("\nADC DMA Example\n");
53
54     // Init ADC
55     // We will read the temperature sensor as fast as possible
56     // and generate a DREQ when a sample goes to the FIFO
57     adc_init();
58     adc_set_temp_sensor_enabled(true);
59     adc_select_input (ADC_INPUT_TEMPSENSOR);
60     adc_fifo_setup (true, true, 1, false, false);
61     adc_set_clkdiv(0);
62
63     // Init DMA
64     dma_chan = dma_claim_unused_channel(true);
65     dma_sniffer_enable(dma_chan, 0xf, false);
66     dma_channel_config c = dma_channel_get_default_config(dma_chan);
67     channel_config_set_transfer_data_size(&c, DMA_SIZE_16);
68     channel_config_set_read_increment(&c, false);
69     channel_config_set_write_increment(&c, true);
70     channel_config_set_dreq(&c, DREQ_ADC);
71     channel_config_set_sniff_enable(&c, true);
72
73     dma_channel_configure(
74         dma_chan,
75         &c,
76         NULL,           // Dont provide a write address yet
77         &adc_hw->fifo,   // Read address (only need to set this once)
78         N_SAMPLES,      // Transfer N_SAMPLES values
79         false           // Dont start yet
80     );
81
82     // DMA will raise IRQ0 when the channel finishes to fill the buffer
83     dma_channel_set_irq0_enabled(dma_chan, true);
84     irq_set_exclusive_handler(DMA_IRQ_0, dma_irq_handler);
85     irq_set_enabled(DMA_IRQ_0, true);

```

```

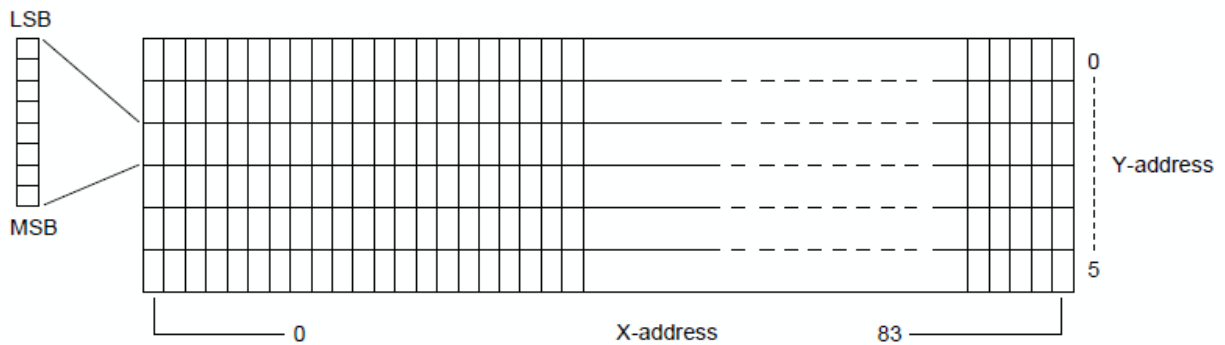
86
87
88 // Start transfer to first buffer
89 dma_hw->sniff_data = 0;
90 dma_channel_set_write_addr(dma_chan, buffer[iBuf], true);
91
92 // Start the ADC
93 adc_run(true);
94
95 // Main loop
96 while (1) {
97     // Make sure last transfer finished
98     dma_channel_wait_for_finish_blocking(dma_chan);
99     uint32_t finished = finishedXfer[iBuf];
100
101     // Get the sum of the samples
102     uint32_t sum = dma_hw->sniff_data;
103
104     // Switch buffers
105     iBuf = 1-iBuf;
106
107     // Set up and start DMA transfer to other buffer
108     dma_hw->sniff_data = 0;
109     dma_channel_set_write_addr(dma_chan, buffer[iBuf], true);
110     adc_run(true);
111
112     // At this point we can process buffer 1-iBuf, we will just
113     // calculate and print average temperature
114     float tempSum = sum * conversionFactor;
115     float tempC = 27.0f - (tempSum/N_SAMPLES - 0.706f) / 0.001721f;
116     printf("Temperature: %.2f ", tempC);
117     uint32_t printed = to_ms_since_boot(get_absolute_time());
118
119     // Show when the transfer finished and when we finished printing
120     printf ("Finished transfer: %u Finished printing: %u\n", finished, printed);
121 }
122 }

```

Sending Data to a SPI LCD Display using DMA

The display used in this example is a Nokia 5110 monochromatic LCD display with 84x48 resolution. The controller chip is a PCD8544 that has an SPI interface.

I will not go in all the details on the PCD8544, the point of interest here is that it has inside a display memory that we will write to using SPI. This memory has 504 bytes, each of them associated with 8 vertical pixels:

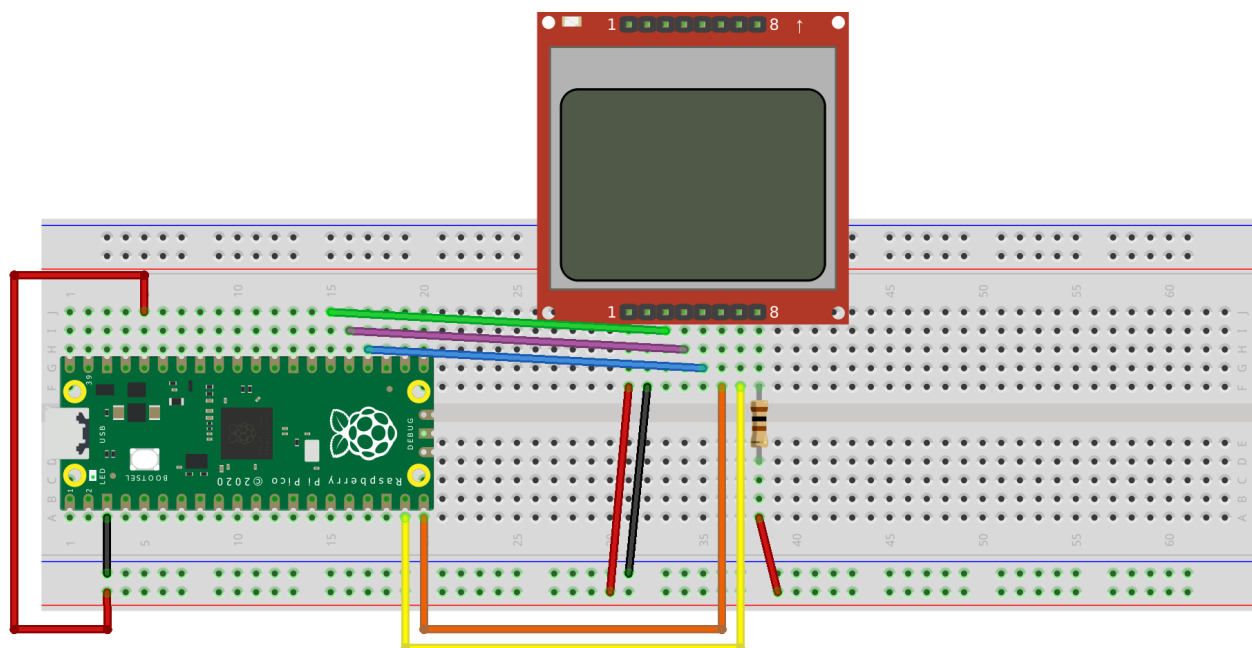


PCD8544 memory

To flex our DMA muscles, we will divide the screen in three horizontal strips and use DMA chaining to get each strip from a separate area of memory.

This is an example where, in the data channel, the write address (the peripheral FIFO) is fixed and the read address is incremented. The transfer count is the number of bytes in the screen strip.

The circuit used in this example is this:



fritzing

Connecting a Nokia 5110 display to the Pi Pico

Here is the code:

Sending Data to a SPI LCD Display using DMA

```
1  /**
2   * @file spidma.c
3   * @author Daniel Quadros
4   * @brief Example of using DMA with SPI in the RP2040
5   *        to drive a Nokia 5110 display
6   * @version 0.1
7   * @date 2022-09-07
8   *
9   * @copyright Copyright (c) 2022, Daniel Quadros
10  *
11  */
12
13  #include <stdio.h>
14  #include <string.h>
15  #include <stdlib.h>
16
17  #include "pico/stdlib.h"
18  #include "hardware/spi.h"
19  #include "hardware/dma.h"
20
21  // Display connections
22  #define PIN_SCE 20
23  #define PIN_RESET 19
24  #define PIN_DC 18
25  #define PIN_SDIN 15
26  #define PIN_SCLK 14
27
28  // Data/Command selection
29  #define LCD_CMD 0
30  #define LCD_DAT 1
31
32  // Screen size
33  #define LCD_DX 84
34  #define LCD_DY 48
35
36  // Display init cmds
37  uint8_t lcdInit[] = { 0x21, 0xB0, 0x04, 0x15, 0x20, 0x0C };
38
39  // Put display pointer in home position
40  uint8_t lcdHome[] = { 0x40, 0x80 };
```

```

41
42 // Each byte in the display memory controls 8 vertical pixels
43 // We are going to divide the display in three horizontal strips:
44 //   Top      8 pixels high
45 //   Main    32 pixels high
46 //   Bottom   8 pixels high
47 uint8_t topScreen[2][LCD_DX];
48 uint8_t mainScreen[2][LCD_DX*4];
49 uint8_t bottomScreen[2][LCD_DX];
50 int screenDMA = 0; // main screen programmed in DMA
51
52 // SPI Configuration
53 #define SPI_ID spi1
54 #define BAUD_RATE 4000000 // 4 MHz
55 #define DATA_BITS 8
56
57 // DMA channel numbers
58 int dma_chan_data;
59 int dma_chan_ctrl;
60
61 // Flag to signal end of screen update
62 volatile bool screenUpdated = true;
63
64 // Control blocks for transferring screen data
65 // We will change the data pointers as needed
66 struct {uint32_t len; const char *data;} control_blocks[] = {
67     {LCD_DX, NULL},
68     {LCD_DX*4, NULL},
69     {LCD_DX, NULL},
70     {0, NULL} // Null trigger to end chain.
71 };
72
73 // This routine will run when the data DMA gets a null trigger
74 void dma_irq_handler() {
75     // Clear the interrupt request.
76     dma_hw->ints0 = 1u << dma_chan_data;
77     // Set flag to indicate end
78     screenUpdated = true;
79 }
80
81 // Init screen buffers
82 void initStrips() {
83     // Horizontal Lines

```

```

84     for (int i = 0; i < LCD_DX; i++) {
85         topScreen[0][i] = 0x55;
86         bottomScreen[0][i] = 0x66;
87     }
88     // Simple Patterns
89     for (int i = 0; i < LCD_DX; i+=2) {
90         topScreen[1][i] = 0x63;
91         topScreen[1][i+1] = 0x63;
92         bottomScreen[1][i] = 0x7F;
93         bottomScreen[1][i+1] = 0x41;
94     }
95     // Main screen is already with zeros
96 }
97
98 // Init DMA
99 void initDMA() {
100     // Get two channels
101     dma_chan_data = dma_claim_unused_channel(true);
102     dma_chan_ctrl = dma_claim_unused_channel(true);
103
104     // Set up control channel
105     dma_channel_config c = dma_channel_get_default_config(dma_chan_ctrl);
106     channel_config_set_transfer_data_size(&c, DMA_SIZE_32);
107     channel_config_set_read_increment(&c, true);
108     channel_config_set_write_increment(&c, true);
109     channel_config_set_ring(&c, true, 3); // 1 << 3 byte boundary on write ptr
110     dma_channel_configure(
111         dma_chan_ctrl,
112         &c,
113         &dma_hw->ch[dma_chan_data].al3_transfer_count,
114         &control_blocks[0],
115         2,
116         false // Dont start yet.
117     );
118
119     // Set up data channel
120     c = dma_channel_get_default_config(dma_chan_data);
121     channel_config_set_transfer_data_size(&c, DMA_SIZE_8);
122     channel_config_set_dreq(&c, spi_get_dreq(SPI_ID, true));
123     channel_config_set_chain_to(&c, dma_chan_ctrl);
124     channel_config_set_irq_quiet(&c, true);
125     dma_channel_configure(
126         dma_chan_data,

```

```
127     &c,
128     &spi_get_hw(SPI_ID)->dr,
129     NULL,    // Initial read address and transfer count
130     0,       // are unimportant
131     false    // Dont start yet.
132 );
133
134 // DMA will raise IRQ0 when it gets a null trigger
135 dma_channel_set_irq0_enabled(dma_chan_data, true);
136 irq_set_exclusive_handler(DMA_IRQ_0, dma_irq_handler);
137 irq_set_enabled(DMA_IRQ_0, true);
138 }
139
140 // Init Display
141 void displayInit() {
142     // Configure GPIO pins
143     gpio_init(PIN_SCE);
144     gpio_set_dir(PIN_SCE, true);
145     gpio_put(PIN_SCE, true);
146     gpio_init(PIN_RESET);
147     gpio_set_dir(PIN_RESET, true);
148     gpio_put(PIN_RESET, true);
149     gpio_init(PIN_DC);
150     gpio_set_dir(PIN_DC, true);
151     gpio_put(PIN_DC, true);
152
153     // Set up SPI
154     uint baud = spi_init (SPI_ID, BAUD_RATE);
155     printf ("SPI @ %u Hz\n", baud);
156     spi_set_format (SPI_ID, DATA_BITS, SPI_CPOL_1, SPI_CPHA_1,
157                    SPI_MSB_FIRST);
158
159     // Set up the SPI pins
160     gpio_set_function(PIN_SCLK, GPIO_FUNC_SPI);
161     gpio_set_function(PIN_SDIN, GPIO_FUNC_SPI);
162
163     // Reset the display controller
164     gpio_put(PIN_RESET, false);
165     sleep_ms(100);
166     gpio_put(PIN_RESET, true);
167
168     // Initialize the display controller
169     // We will not use DMA for this
```

```
170     gpio_put(PIN_SCE, false); // leave it selected
171     gpio_put(PIN_DC, false);
172     spi_write_blocking(SPI_ID, lcdInit, sizeof(lcdInit));
173     gpio_put(PIN_DC, true);
174 }
175
176 // Refresh the screen
177 void displayRefresh(int top, int bottom) {
178     // Make sure previous refresh is finished
179     while (!screenUpdated) {
180         tight_loop_contents();
181     }
182     screenUpdated = false;
183
184     // Switch buffer
185     screenDMA = 1 - screenDMA;
186
187     // Update data address in control block
188     control_blocks[0].data = topScreen[top];
189     control_blocks[1].data = mainScreen[screenDMA];
190     control_blocks[2].data = bottomScreen[bottom];
191
192     // Position data pointer at start of memory
193     // (also not using DMA for this)
194     gpio_put(PIN_DC, false);
195     spi_write_blocking(SPI_ID, lcdHome, sizeof(lcdHome));
196     gpio_put(PIN_DC, true);
197
198     // Start DMA
199     // Control channel will set the data channel transfers
200     dma_channel_set_read_addr(dma_chan_ctrl, &control_blocks[0],
201         true);
202 }
203
204 // Draw the next frame
205 const uint8_t masks[] = { 0xC0, 0xF0, 0x0C, 0x0F };
206 void drawFrame() {
207     int s = 1 - screenDMA;
208
209     // Copy previous screen
210     memcpy(mainScreen[s], mainScreen[screenDMA],
211         sizeof(mainScreen[0]));
212 }
```

```
213 // Erase a random rectangle
214 int n = (rand() % 16) + 2;
215 int x = rand() % (LCD_DX - n);
216 int y = rand() % 4;
217 uint8_t mask = masks[rand() % 4];
218 for (int i = 0; i < n ; i ++) {
219     mainScreen[s][LCD_DX*y+x+i] &= mask;
220 }
221
222 // Draw a random rectangle
223 n = (rand() % 16) + 2;
224 x = rand() % (LCD_DX - n);
225 y = rand() % 4;
226 mask = masks[rand() % 4];
227 for (int i = 0; i < n ; i ++) {
228     mainScreen[s][LCD_DX*y+x+i] |= mask;
229 }
230 }
231
232 // Main Program
233 int main() {
234     // Init screen
235     initStrips();
236     initDMA();
237     displayInit();
238     displayRefresh(0, 0);
239
240     // Main loop
241     int frameCounter = 0;
242     int border = 0;
243     while (1) {
244         sleep_ms(100);
245         drawFrame();
246         displayRefresh(border & 1, (border & 2) >> 1);
247         if (++frameCounter == 100) {
248             // Change borders from time to time
249             frameCounter = 0;
250             border = (border + 1) & 3;
251         }
252     }
253 }
```

Some highlights on the DMA usage in this example:

- In `initDMA()` we allocate two DMA channels. `dma_chan_data` will transfer the data from the RP2040 memory to the SPI peripheral. `dma_chan_ctrl` will program `dma_chan_data` by transferring the read address and transfer count from a control block.
- In `displayRefresh()` we fill in the addresses in the control blocks.
- The end of the control block is a *null trigger*. We set up an interrupt to occur when the null trigger is reached.
- Alas, when the interrupt occurs data is still in the SPI FIFO for transmission, so we cannot turn off the select signal of the display at this time. To simplify things, I just left it on.

Clock Generation, Timer, Watchdog and RTC

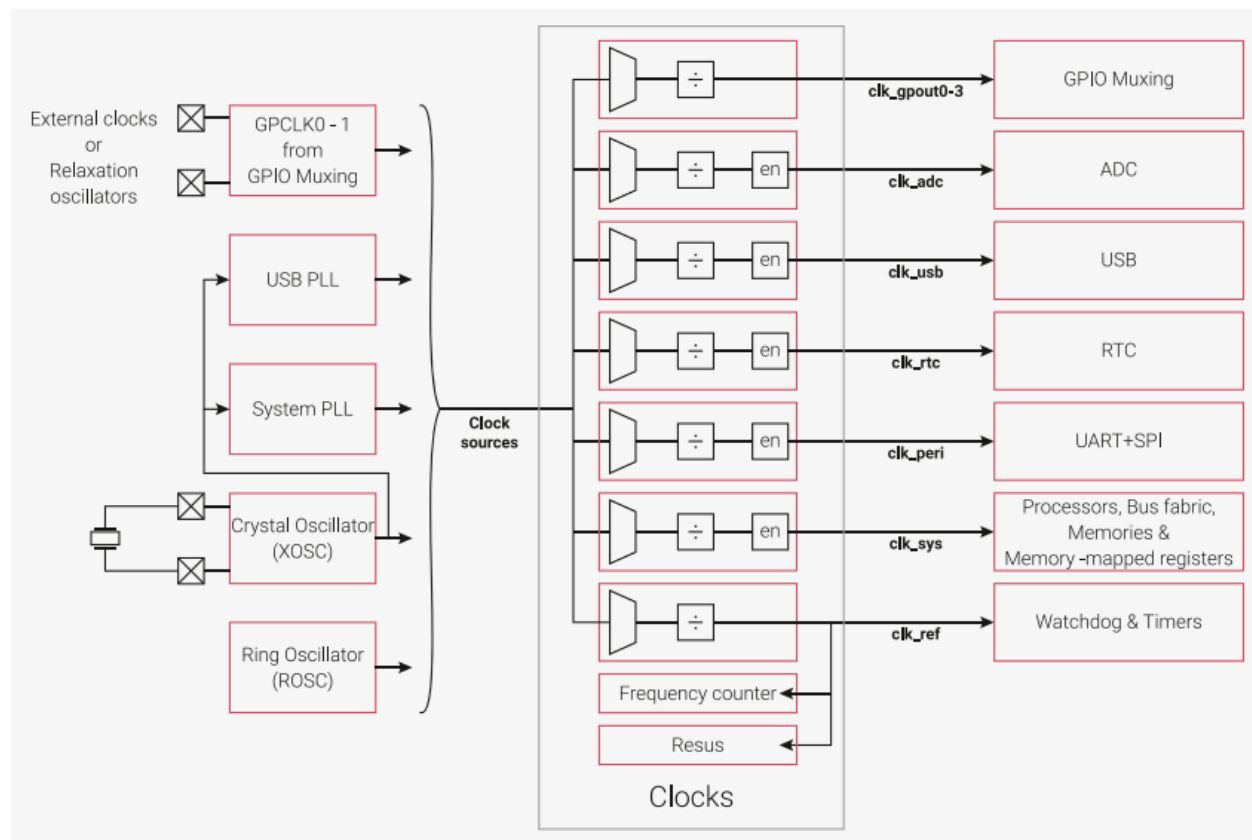
In this chapter we are going to see the clock sources available to the RP2040 and three peripherals that use them:

- **Timer** provides a 64 bit microsecond counter that can be used to generate interrupts.
- **Watchdog** restarts the RP2040 if it is not periodically reset by software (used to recover from software malfunction).
- **RTC** (Real Time Clock) keeps time in day, month, year, hour, minute and second format (as long as the RP2040 is powered). Can generate an interrupt at a certain date and time.

Clock Generation

Overview

The RP2040 has a very flexible clock subsystem, with many options of **clock sources**. The actual clock used by the processors and peripheral comes from a **clock generator** that selects one of the sources and divide it. Many clock generators can be disabled in SLEEP mode to save power.



The Clock Subsystem

This table shows what output of the clock generators drive each subsystem:

Subsystem	Clock	Usual Source
Processor	f_sys	System PLL
I2C	f_sys	System PLL
USB	clk_usb	USB PLL
ADC	clk_adc	USB PLL
RTC	clk_rtc	XOSC
Timers	clk_ref	XOSC
Watchdog	clk_ref	XOSC
SPI	clk_peri	System PLL or XOSC
UART	clk_peri	System PLL or XOSC

ROSC - Ring Oscillator

The ROSC is an on-chip source that requires no external component and uses little power. But it is not accurate: the typical value is 6MHz, but can change due to fabrication on environment changes. It is expected to be in the range of 4 to 8 MHz, but is only guaranteed to be between 1.8 and 12 MHz.

This source starts at power up and is used in the initial boot stages. It can be powered down, if first you switch its users to another source.

Should you want to use the ROSC as your main clock source, the RP2040 datasheet has some tips in how to mitigate its wide range.

XOSC - Crystal Oscillator

The XOSC can be used to get a precise and stable clock and is the preferred choice. It requires an external crystal in the range 1 to 15MHz (12MHz is the value in the reference design and in the Raspberry Pi Pico). This clock can be fed into the PLLs to generate higher frequencies.

In typical use, the XOSC will drive the clock for the timer, watchdog and RTC (`clk_ref` and `clk_rtc`).

External Clocks

Up to three external clocks can be connected to pins GPIO0, GPIO1 and XIN. This inputs are limited to 50MHz but can be fed into the PLLs to generate higher frequencies.

This option is interesting if your board has a precise clock signal that can be used, as it saves the cost of an external crystal.

PLLs

The PLLs (*Phase Locked Loops*) in the RP2040 can multiply the frequency of the XOSC (or an external clock at XIN) to generate a faster clock.

There are two PLLs in the RP2040. The **USB PLL** is typically used to generate the 48MHz clock needed for the USB and ADC. The **System PLL** is used to generate `clk_sys`.

Depending on the needs for UART and SPI, `clk_peri` will be driven from XOSC or System PLL.

The System PLL will usually run at 125MHz, but can be increased to overclock the processor, or reduced to lower power consumption.

Clock Output

Up to four clocks can be outputted in GPIO pins. This can be used to provide a clock signal to other devices or for testing purposes.

Only GPIOs 21, 23, 24 and 25 can be used for clock output. In the Raspberry Pi Pico only GPIO 21 is available (GPIO25 is connected to the LED, GPIO 23 and 24 are not brought to the connector).

Frequency Counter

The frequency counter can be used to measure the frequency of a source by counting the clock edges seen over a test interval. The interval is defined by counting cycles of `clk_ref` (which should have a known and stable frequency).

There are sixteen options of interval. A short interval means that the test will be fast, but imprecise.

Option	Test Interval	Accuracy
0	1 μ s	2048 kHz
1	2 μ s	1024 kHz
2	4 μ s	512 kHz
3	8 μ s	256 kHz
4	16 μ s	128 kHz
5	32 μ s	64 kHz
6	64 μ s	32 kHz
7	125 μ s	16 kHz
8	250 μ s	8 kHz
9	500 μ s	4 kHz
10	1 ms	2 kHz
11	2 ms	1 kHz
12	4 ms	500 Hz
13	8 ms	250 Hz
14	16 ms	125 Hz
15	32 ms	62.5 Hz

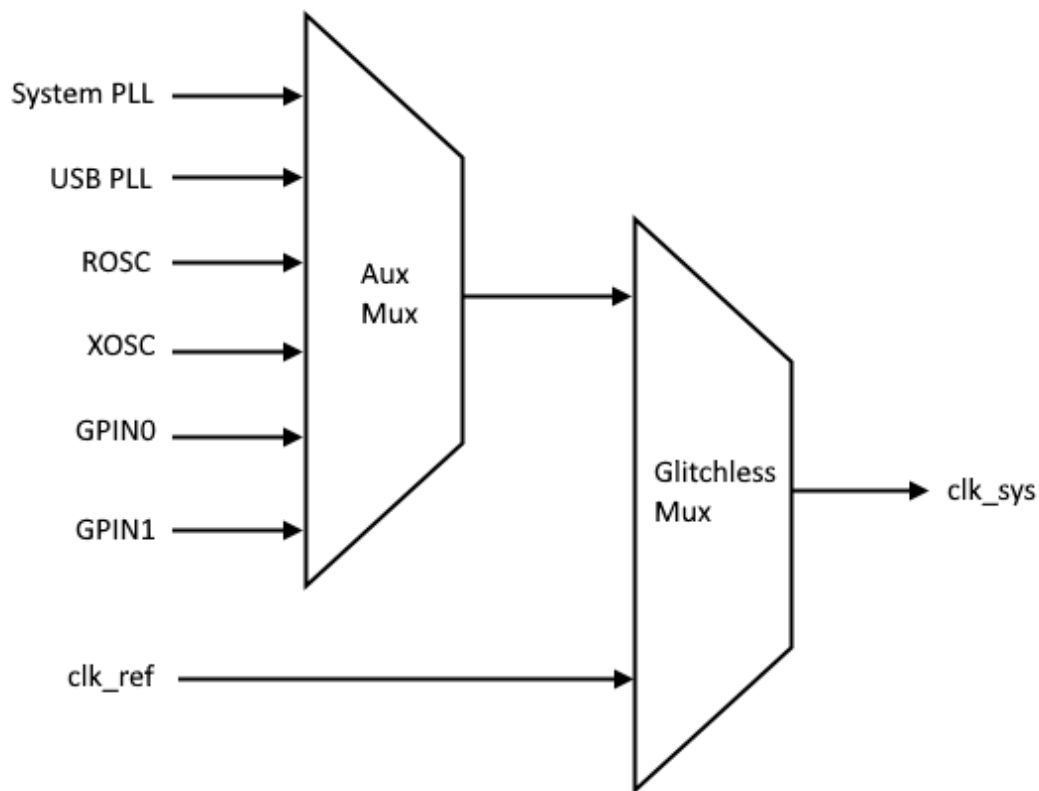
Clock Generator Multiplexers

The selection of the source for a clock generator is made by one or two *multiplexers*. A multiplexer is a circuit that has two sets of inputs (the sources and the source selection) and one output; the output receives the signal in the selected source.

All clock generators have what the datasheet and SDK calls an **auxiliary (aux) mux**. This mux will *glitch* when the source is changed (that is, for an instant, the output will not be equal to the previous nor the new signal). This glitch can cause problems to the circuit that uses the clock.

For sources that have only an aux mux, the clock should be stopped (disabled) while the source is changed.

The generators for `clk_ref` (used for timer and watchdog) and `clk_sys` (used for the processors) have also a **glitchless mux**, because this clocks cannot be stopped. This second mux is after the aux mux, as shown bellow for the `clk_sys` generator:



System Clock Muxes

Suppose you're running `clk_sys` from XOSC and want to change it to System PLL. If you change directly in the aux mux, a glitch can stop (or confuse) the processors. So, first you change the source to `clk_ref`, using the glitchless mux. Then you change the source in the aux mux. Finally you change again the glitchless mux to select the output of the aux mux.

There are also other precautions when changing the clock source or its frequency, like waiting for the output to stabilize. Thankfully the SDK has a function that to do this the right way.

Selected SDK Functions

These functions are in the library `hardware_clocks`.

The `clock_index` enum is used to select a clock:

- `clk_gpout0`, `clk_gpout1`, `clk_gpout2` and `clk_gpout3` are the clocks that can be outputted through GPIO pins.
- `clk_ref` is the clock used in the timer and watchdog
- `clk_sys` is the clock for the processor and I2C
- `clk_peri` is the clock for SPI and UART
- `clk_usb` is the clock for USB

- `clk_adc` is the clock for ADC
- `clk_rtc` is the clock for RTC

```
void clocks_init (void)
```

This function initializes the library and must be called before the other functions.

```
bool clock_configure (enum clock_index clk_index, uint32_t src, uint32_t auxsrc, uint32_t src_freq, uint32_t freq)
```

Configures the `clk_index` clock to operate at frequency `freq` using sources `src` and `auxsrc`. If the generator for the clock has only an aux mux, the source can be passed in `src` and `aux_src` can be zero. If the generator for the clock has two muxes, `src` applies to the glitchless and `auxsrc` to the aux mux. The values for these parameters can be found in the official documentation. `src_freq` is the frequency of the source and is used by the function when waiting for the output of the muxes to stabilize.

The function returns false if the request cannot be fulfilled.

```
void clock_stop (enum clock_index clk_index)
```

Stops a clock. Used for power saving, make sure that you are not using the clock that you are stopping.

```
uint32_t clock_get_hz (enum clock_index clk_index)
```

Return the current frequency (in hertz) for a clock. The returned value will be from the most recent `clock_configure()` or `clock_set_reported_hz()` call.

```
void clock_set_reported_hz (enum clock_index clk_index, uint hz)
```

Set the current frequency returned by `clock_get_hz()` but does not change its frequency. This only makes sense if the clock frequency was changed from outside `clock_configure()`.

```
uint32_t frequency_count_khz (uint src)
```

Uses the Frequency Counter to measure a clock's frequency. Uses a test interval of 2us with a result accuracy of +/- 1KHz.

```
void clock_gpio_init (uint gpio, uint src, uint div)
```

Configure a clock to be outputted in a GPIO pin. `gpio` must be 21, 23, 24 or 25. `src` is the clock source and `div` the divisor to be applied.

```
bool clock_configure_gpin (enum clock_index clk_index, uint gpio, uint32_t src_freq, uint32_t freq)
```

Configure a clock to use as source a GPIO pin. `gpio` must be 20 or 22. `src_freq` is the input frequency and `freq` is the desired frequency for the clock.

The function returns false if the request cannot be fulfilled.

Example

In this example (based on the SDK clock examples) we use the frequency counter to measure some of the clocks. Then we change the source for clock_sys to USB PLL (dropping the processor clock down to 48MHz) and do the measuring again. This example also outputs the ROsc clock, divided by 10, through GPIO 21.

Clocks Example

```

1  /**
2   * @file clocksdemo.c
3   * @author Daniel Quadros
4   * @brief Example of using the Clocks API
5   *        Based on the hello_48MHz and hello_gpout SDK examples
6   * @version 0.1
7   * @date 2022-07-14
8   *
9   * @copyright Copyright (c) 2022, Daniel Quadros
10  *
11  */
12
13  #include <stdio.h>
14  #include "pico/stdlib.h"
15  #include "hardware/pll.h"
16  #include "hardware/clocks.h"
17  #include "hardware/structs/pll.h"
18  #include "hardware/structs/clocks.h"
19
20  // Use the frequency counter to measure the various clocks
21  void measure_freqs(void) {
22      uint f_pll_sys = frequency_count_khz(CLOCKS_FC0_SRC_VALUE_PLL_SYS_CLKSRC_PRIMARY\
23  );
24      uint f_pll_usb = frequency_count_khz(CLOCKS_FC0_SRC_VALUE_PLL_USB_CLKSRC_PRIMARY\
25  );
26      uint f_rosc = frequency_count_khz(CLOCKS_FC0_SRC_VALUE_ROSC_CLKSRC);
27      uint f_clk_sys = frequency_count_khz(CLOCKS_FC0_SRC_VALUE_CLK_SYS);
28      uint f_clk_peri = frequency_count_khz(CLOCKS_FC0_SRC_VALUE_CLK_PERI);
29      uint f_clk_usb = frequency_count_khz(CLOCKS_FC0_SRC_VALUE_CLK_USB);
30      uint f_clk_adc = frequency_count_khz(CLOCKS_FC0_SRC_VALUE_CLK_ADC);
31      uint f_clk_rtc = frequency_count_khz(CLOCKS_FC0_SRC_VALUE_CLK_RTC);
32
33      printf("pll_sys  = %dkHz\n", f_pll_sys);
34      printf("pll_usb  = %dkHz\n", f_pll_usb);
35      printf("rosc    = %dkHz\n", f_rosc);

```

```
36     printf("clk_sys  = %dkHz\n", f_clk_sys);
37     printf("clk_peri = %dkHz\n", f_clk_peri);
38     printf("clk_usb  = %dkHz\n", f_clk_usb);
39     printf("clk_adc  = %dkHz\n", f_clk_adc);
40     printf("clk_rtc  = %dkHz\n", f_clk_rtc);
41
42     stdio_flush(); // make sure output is sent before continuing
43 }
44
45 int main() {
46     stdio_init_all();
47     while (!stdio_usb_connected()) {
48         sleep_ms(100);
49     }
50
51     printf("Clocks Example\n\n");
52
53     // Output ROSC/10 through GPIO21
54     clock_gpio_init(21, CLOCKS_CLK_GPOUT0_CTRL_AUXSRC_VALUE_ROSC_CLKSRC, 10);
55     printf("ROSC/10 now at GPIO21\n\n");
56
57     // Measure frequencies
58     measure_freqs();
59
60     // Change the source of clk_sys to the USB PLL
61     // which has a source frequency of 48MHz
62     clock_configure(clk_sys,
63                   CLOCKS_CLK_SYS_CTRL_SRC_VALUE_CLKSRC_CLK_SYS_AUX,
64                   CLOCKS_CLK_SYS_CTRL_AUXSRC_VALUE_CLKSRC_PLL_USB,
65                   48 * MHZ,
66                   48 * MHZ);
67
68     // No need for System PLL now
69     pll_deinit(pll_sys);
70
71     // In case stdio is through UART, we need to correct clk_peri and reinit stdio
72     clock_configure(clk_peri,
73                   0,
74                   CLOCKS_CLK_PERI_CTRL_AUXSRC_VALUE_CLK_SYS,
75                   48 * MHZ,
76                   48 * MHZ);
77     stdio_init_all();
78 }
```

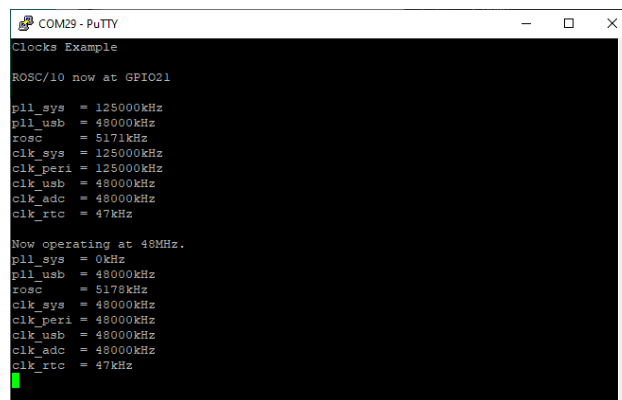


```

79     printf("\nNow operating at 48MHz.\n");
80     measure_fregs();
81
82     // That's all
83     while (true) {
84         sleep_ms(100);
85     }
86
87     return 0;
88 }

```

Here is a sample output:



```

COM29 - PuTTY
Clocks Example

ROSC/10 now at GPIO21

pll_sys = 125000kHz
pll_usb = 48000kHz
rosc    = 5171kHz
clk_sys = 125000kHz
clk_peri = 125000kHz
clk_usb = 48000kHz
clk_adc = 48000kHz
clk_rtc = 47kHz

Now operating at 48MHz.
pll_sys = 0kHz
pll_usb = 48000kHz
rosc    = 5178kHz
clk_sys = 48000kHz
clk_peri = 48000kHz
clk_usb = 48000kHz
clk_adc = 48000kHz
clk_rtc = 47kHz

```

Sample output of the clocks example

Some observations:

- ROSC was around 5.2MHz, somewhat afar from the 6MHz “typical value”, but inside the 4 to 8 MHz “expected range”.
- After turning the System PLL off the frequency measured was zero.
- The nominal frequency for `clk_rtc` is 46875Hz, the `frequency_count_khz()` routine is not very appropriate for measuring it as it has an accuracy of +/- 1KHz.

Timer

The timer peripheral is a 64-bit microsecond counter that can be read and used for up to four alarms.

The time base for the timer is generated by the Watchdog from the the reference clock (that is normally derived from XOSC).

The 64-bit counter can count for thousand of years before overflowing. For all practical uses it will continuously increase during the execution of the software (what is called *monotonic*). This simplifies tasks like computing a elapsed time and waiting for a future time.

While the timer is read through two 32-bit registers, you do not need to worry about one register changing while you are reading the other, as long as you read the low part first. When the low part is read, the high part is stored (latched) and used for the following reading of the high part.

The four alarms generates interrupts on a match of the lower 32-bits of the counter to the alarm value. Since 2^{32} microseconds is about 72 minutes, the alarms should be used for times between tens of microseconds to one hour. Times shorter than ten microseconds will have a significant imprecision. The PIO can be used for small times, as it can run at the system clock. For longer times the RTC can be used.

Selected SDK Functions

The timer functions are in `hardware_timer`. There is a simple control of the use of the alarms.

```
static uint32_t time_us_32 (void)
```

Returns the lower 32 bits of the timer's counter.

```
uint64_t time_us_64 (void)
```

Returns the full 64 bits of the timer's counter.

```
void busy_wait_us_32 (uint32_t delay_us)
```

This routine will return after `delay_us` microseconds. The processor will be in a loop while waiting.

```
void busy_wait_us (uint64_t delay_us)
```

This routine will return after `delay_us` microseconds. The processor will be in a loop while waiting.

```
void busy_wait_ms (uint32_t delay_ms)
```

This routine will return after `delay_ms` milliseconds. The processor will be in a loop while waiting.

```
void busy_wait_until (absolute_time_t t)
```

This routine will return when the counter reaches `t`. The processor will be in a loop while waiting.

```
static bool time_reached (absolute_time_t t)
```

Returns true if the counter is equal or greater `t`.

```
void hardware_alarm_claim (uint alarm_num)
```

Claims the use of an alarm. If the alarm is claimed (in use), the software is stopped by an assert.

```
void hardware_alarm_unclaim (uint alarm_num)
```

Frees an alarm for another use.

```
bool hardware_alarm_is_claimed (uint alarm_num)
```

Returns true is an alarm is in use (claimed).

```
void hardware_alarm_set_callback (uint alarm_num, hardware_alarm_callback_t callback)
```

Sets the routine that will be called when an alarm expires and enables the interrupt. NULL disables the interrupt.

```
bool hardware_alarm_set_target (uint alarm_num, absolute_time_t t)
```

Sets the time when the alarm will expire. Returns true if t is equal or greater the timer's counter (it is "in the past").

```
void hardware_alarm_cancel (uint alarm_num)
```

Cancel an alarm.

pico_time Selected Functions

The functions in the previous section are low level and not particularly useful. The `pico_time` routines (that are part of the `pico_stdlib` library) offers a higher level and more friendly routines. It is divided in four modules: `timestamp`, `sleep`, `alarm` and `repeating_timer`.

timestamp

Instants in time (*timestamps*) are represented by the type `absolute_time_t`. This type hides the actual type used (spoiler: its `uint64_t`) and distinguishes timestamps from other integers (like time intervals). Timestamps are counted from "boot" (actually from the start of the hardware timer, but you should treat it just as an arbitrary reference).

```
static uint64_t to_us_since_boot (absolute_time_t t)
```

```
static uint32_t to_ms_since_boot (absolute_time_t t)
```

This routines convert a timestamp into the number of microseconds and milliseconds.

```
static absolute_time_t get_absolute_time (void)
```

Returns a timestamp that corresponds to "now".

```
static void update_us_since_boot (absolute_time_t *t, uint64_t us_since_boot)
```

Converts a count of microseconds since boot (for example, the current value of the timer) into a timestamp.

```
static absolute_time_t delayed_by_us (const absolute_time_t t, uint64_t us)
```

```
static absolute_time_t delayed_by_ms (const absolute_time_t t, uint32_t ms)
```

Add a number of microseconds or milliseconds to a timestamp.

sleep

The sleep functions delay execution in a low power state.

```
void sleep_until (absolute_time_t target)
```

Sleep until the specified timestamp.

```
void sleep_us (uint64_t us)
```

```
void sleep_ms (uint32_t ms)
```

Sleep for a number of microseconds or milliseconds.

alarm

This routines build upon the Timer alarms, by creating *alarm pools* for each timer alarm. Each alarm pool can have multiple concurrent alarms.

The *default pool* uses timer alarm number 3 and supports up to 16 alarms.

```
void alarm_pool_init_default (void)
```

Initializes the default alarm pool.

```
alarm_pool_t * alarm_pool_get_default (void)
```

Returns a pointer to the default alarm pool.

```
alarm_pool_t * alarm_pool_create (uint hardware_alarm_num, uint max_timers)
```

Creates an alarm pool, using `hardware_alarm_num` timer alarm and supporting up to `max_timers` alarms

```
uint alarm_pool_hardware_alarm_num (alarm_pool_t *pool)
```

Returns the number of the timer alarm used by an alarm pool.

```
void alarm_pool_destroy (alarm_pool_t *pool)
```

Destroy an alarm pool, freeing the associate timer alarm.

```
alarm_id_t alarm_pool_add_alarm_at (alarm_pool_t *pool, absolute_time_t time, alarm_-  
callback_t callback, void *user_data, bool fire_if_past)
```

```
static alarm_id_t alarm_pool_add_alarm_in_us (alarm_pool_t *pool, uint64_t us, alarm_-  
callback_t callback, void *user_data, bool fire_if_past)
```

```
static alarm_id_t alarm_pool_add_alarm_in_ms (alarm_pool_t *pool, uint32_t ms, alarm_-  
callback_t callback, void *user_data, bool fire_if_past)
```

This routines add an alarm to an alarm pool. The callback routine will be called when the alarm fires, receiving `user_data` as a parameter. The callback will be called from the timer interrupt routine, normally in core 0. If the callback returns a non-zero value, the alarm will be re-triggered for `value` microseconds after the current timestamp (if value is positive) or `value` microseconds after the previous target (if value is negative).

If `fire_if_past` is true, the alarm will fire immediately if the target time has already passed.

The routines returns an id that identifies the alarm in the alarm pool. The id will be -1 if there is no space in the alarm pool.

```
bool alarm_pool_cancel_alarm (alarm_pool_t *pool, alarm_id_t alarm_id)
```

Cancels an alarm.

repeating_timer

This routines are similar to the `alarm_` functions, but the library remembers the initial delay through a `repeating_timer_t` structure.

```
bool alarm_pool_add_repeating_timer_us (alarm_pool_t *pool, int64_t delay_us,
repeating_timer_callback_t callback, void *user_data, repeating_timer_t *out)
```

```
static bool alarm_pool_add_repeating_timer_ms (alarm_pool_t *pool, int32_t delay_ms,
repeating_timer_callback_t callback, void *user_data, repeating_timer_t *out)
```

This routines add a repeating alarm in an alarm pool. `callback` will be called at every delay interval, until it returns false. If delay is positive, it will be counted from the actual timestamp of the return of the callback, if its negative it will be counted from the previous target.

This functions return false if there is no space in the alarm pool.

```
static bool add_repeating_timer_us (int64_t delay_us, repeating_timer_callback_t
callback, void *user_data, repeating_timer_t *out)
```

```
static bool add_repeating_timer_ms (int32_t delay_ms, repeating_timer_callback_t
callback, void *user_data, repeating_timer_t *out)
```

Same as `alarm_pool_add_repeating_` but using the default alarm pool.

```
bool cancel_repeating_timer (repeating_timer_t *timer)
```

Cancels a repeating timer.

Example

The main objective of this example is to show the use of low level timer functions, but it also uses a few of the `pico_time` routines. The other examples in this book uses only `pico_time` when there is a need for timer functions.

Timer Example

```
1  /**
2   * @file ctimerdemo.c
3   * @author Daniel Quadros
4   * @brief Example of using the Timer
5   * @version 0.1
6   * @date 2022-07-14
7   *
8   * @copyright Copyright (c) 2022, Daniel Quadros
9   *
10  */
11
12  #include <stdio.h>
13  #include <stdlib.h>
14  #include "pico/stdlib.h"
```

```

15  #include "hardware/timer.h"
16
17  #define ALARM_NO 1
18
19  static volatile bool fired;
20
21  // Alarm callback routine
22  void rotAlarm(uint alarm_num) {
23      printf ("Alarm %d fired\n");
24      fired = true;
25  }
26
27  // Main program
28  int main() {
29      stdio_init_all();
30      while (!stdio_usb_connected()) {
31          sleep_ms(100);
32      }
33
34      printf("Timer Example\n\n");
35
36      // Reading the timer a few times
37      for (int i = 0; i < 5; i++) {
38          printf("Timer: %llu\n", time_us_64());
39          busy_wait_us_32(rand() % 10000);    // wait a random time 0 to 9,999 us
40      }
41      printf ("\n");
42
43      // Set up the alarm
44      hardware_alarm_claim(ALARM_NO);
45      hardware_alarm_set_callback(ALARM_NO, rotAlarm);
46
47      // Wait for the alarm at random times
48      while (true) {
49          fired = false;
50          uint32_t delay = 1000 * (1 + rand() % 30);    // 1 to 30 sconds
51          absolute_time_t now;
52          update_us_since_boot(&now, time_us_64());
53          absolute_time_t target = delayed_by_ms(now, delay);
54          hardware_alarm_set_target(ALARM_NO, target);
55          printf ("Waiting for %llu (delay %us)\n",  to_us_since_boot(target), delay/1\
56 000);
57          while (!fired) {

```

```
58         tight_loop_contents();
59     }
60     printf("Timer: %llu\n\n", time_us_64());
61 }
62
63 return 0;
64 }
```

Watchdog

A watchdog is a common microcontroller feature. Its objective is to put the system in a known state (by resetting it) should the firmware misbehaves and “stuck” at some point of the code.

The watchdog is implemented as a counter that, when enabled, will continually decrements and reset the microcontroller when it reaches zero. To avoid the reset, the software has to re-trigger it periodically before the reset.

The clock for the RP2040’s watchdog is `clk_tick`, the same as for the timer, and it is generated from `clk_ref`. For precision, the `clk_ref` itself should be configured to use the Crystal Oscillator. The SDK initializes the clocks so that `tick` is nominally 1us (assuming a 12MHz crystal).

At the hardware level there are a few details that are abstracted by the SDK functions:

- The re-trigger of the watchdog is done by reloading the counter. The SDK stores internally the value specified when the watchdog is enabled.
- Due to a bug in the hardware, the RP2040 decrements the counter twice at each tick. The SDK functions take this into account.
- The watchdog includes eight 32-bit scratch registers. These registers are cleared at power up or external reset but keep their values in case of a reset triggered by the watchdog. These registers are used by the *Bootrom code* (the code that is in the RP2040 Rom and is executed before anything else).

To make good use of the watchdog you have to choose carefully where you re-trigger it. On one hand you must assure that the watchdog will not trip on normal operation and at the other you want it to reset even if the software is running but not doing some important tasks.

Most softwares will simply re-trigger the watchdog in the main loop. This gives a reasonable protection, but only to bugs that stop the execution of the main loop. Care must also be taken with special situations where the main loop is not execute for some time, the watchdog must be re-trigger periodically at other places in this situations.

Selected SDK Functions

```
void watchdog_enable (uint32_t delay_ms, bool pause_on_debug)
```

Initialize and enable the watchdog. `delay_ms` is the time in milliseconds before the watchdog resets the RP2040. If `pause_on_debug` is true, the watchdog will be disabled when a debugger is stepping through code.

The watchdog counter is 24 bits wide. As it is decremented twice each 1us tick, the maximum value for `delay_ms` is 8388 (a little more than 8 seconds).

Notice that the SDK does not include a `watchdog_disable()` function.

```
void watchdog_update (void)
```

Re-triggers the watchdog, by reloading the value specified in `watchdog_enable()`

```
bool watchdog_caused_reboot (void)
```

Returns true if the watchdog caused a reboot.

```
uint32_t watchdog_get_count (void)
```

Returns the number of microseconds before the watchdog resets the microcontroller.

Example

In this example we first check if the program started from a normal reset or a watchdog reset. Then we enable the watchdog with a 100ms timeout and enter a loop where we sleep a random number of milliseconds. This random number is between 0 and 100, imprecision in the sleep routine and the time spent on `printf()` will cause a watchdog reset after a few seconds.

Watchdog Example

```
1  /**
2   * @file watchdogdemo.c
3   * @author Daniel Quadros
4   * @brief Example of using the Watchdog
5   * @version 0.1
6   * @date 2022-07-14
7   *
8   * @copyright Copyright (c) 2022, Daniel Quadros
9   *
10  */
11
12  #include <stdio.h>
13  #include <stdlib.h>
14  #include "pico/stdlib.h"
15  #include "hardware/watchdog.h"
```



```

16
17 int main() {
18     // Init sdio
19     stdio_init_all();
20     while (!stdio_usb_connected()) {
21         sleep_ms(100);
22     }
23
24     printf("Watchdog Example\n\n");
25     if (watchdog_caused_reboot()) {
26         printf("Rebooted by Watchdog!\n");
27         sleep_ms(500);
28     } else {
29         printf("Clean boot\n");
30     }
31
32     // Enable the watchdog with a 100ms timeout
33     watchdog_enable(100, false);
34
35     // Lets play watchdog roulette!
36     while (true) {
37         printf(".");
38         sleep_ms(rand() % 101); // 0 to 100ms
39         watchdog_update();
40     }
41
42     return 0;
43 }

```

If you using stdio through USB, the reset will abort the communication. You will have to restart it to confirm that it was caused by the watchdog.

RTC

The Real Time Clock makes it easy to maintain the current date and time while the RP2040 is powered.

Its important to remember that the RTC does not generate or compute information. The firmware is responsible for loading a valid date and time; the RTC will update it each second, following the normal time and date sequence (including, partially, leap years).

The RTC updates seven fields each second:

- Year: from 0 to 4095
- Month: from 1 to 12
- Day: from 1 to 28, 29, 30 or 31
- Day of the Week: from 0 to 6
- Hour: from 0 to 23
- Minute: from 0 to 59
- Seconds: from 0 to 59

Again, it is up to the firmware to load valid values. There is no guarantee of what will happen if illegal values are loaded.

There is no association between the day of the week and the date, the RTC will increment the value each day, wrapping from 6 to 0. The SDK adopts a convention that 0 is Sunday.

Years that are multiple of four are considered leap years and February 28 will be followed by February 29 instead of March 1. Notice that the full leap year rule states that year multiple of 100 are not leap unless it is multiple of 400. If you want to use the full rule you need to manually turn off the RTC leap year check in the exceptions to the multiple of 4 rule.

The RTC can work as long as the RP2040 is powered and it has a clock. You can use the SLEEP or DORMANT states (see chapter 4) to stop the processors and reduce the power consumption and power the RP2040 through an external battery circuit to maintain the clock, date and time while the main power source is not available.

The RTC has an alarm that can match on any combination of the seven fields. For example, we can set the alarm to occur at 16:01:23 regardless of the date. The SDK will keep the alarm enabled by default when not all fields are specified, so if you specify just 16:01 it will occur at every second while hour is 16 and minute is 1.

Selected SDK Functions

This functions are in the library `hardware_rtc`.

```
void rtc_init (void)
```

Initializes the RTC, setting up its clock.

```
bool rtc_set_datetime (datetime_t *t)
```

Sets the RTC to the date and time provided. Returns false if date/time invalid.

```
bool rtc_get_datetime (datetime_t *t)
```

Fills a `datetime_t` structure with the current date and time in the RTC. Returns false if RTC is not running.

```
bool rtc_running (void)
```

Returns true if the RTC is running.

```
void rtc_set_alarm (datetime_t *t, rtc_callback_t user_callback)
```

Sets the alarm's date and time. Fields with -1 in t will not be used in the match. user_callback will be called when the alarm is reached. The alarm is enabled.

If any field in t is -1 the alarm is re-enabled before calling user_callback. If that is not what you want, call rtc_disable_alarm() in user_callback.

```
void rtc_enable_alarm (void)
```

Enables the alarm.

```
void rtc_disable_alarm (void)
```

Disables the alarm.

Example

In this example the RTC is programmed with a date and time typed through the standard input. Then the alarm is exercised using random intervals.

RTC Example

```

1  /**
2   * @file rtcdemo.c
3   * @author Daniel Quadros
4   * @brief Example of using the Real Time Clock
5   *        Based on the hello_48MHz and hello_gpout SDK examples
6   * @version 0.1
7   * @date 2022-07-14
8   *
9   * @copyright Copyright (c) 2022, Daniel Quadros
10  *
11  */
12
13  #include <stdio.h>
14  #include <stdlib.h>
15  #include "pico/stdlib.h"
16  #include "pico/util/datetime.h"
17  #include "hardware/rtc.h"
18
19  static volatile bool fired;
20
21  // This routine will be called when the alarm fires
22  static void alarm_callback(void) {
23      datetime_t dt;
24  
```

```
25     // Disable alarm
26     rtc_disable_alarm();
27
28     // Get the current time and convert it to a string
29     rtc_get_datetime(&dt);
30     char datetime_buf[256];
31     char *datetime_str = &datetime_buf[0];
32     datetime_to_str(datetime_str, sizeof(datetime_buf), &dt);
33
34     // Inform alarm fired
35     printf("Alarm fired at %s\n", datetime_str);
36     stdio_flush();
37     fired = true;
38 }
39
40
41 // Main Program
42 int main() {
43     stdio_init_all();
44     while (!stdio_usb_connected()) {
45         sleep_ms(100);
46     }
47
48     printf("RTC Example\n");
49
50     // Initializes the RTC
51     datetime_t dt;
52     rtc_init();
53     while (true) {
54         int dig[14];
55         int n = 0;
56         int c;
57         printf("Enter date and time as MMDDYYYYHHMMSS\n");
58         while (n < 14) {
59             c = getchar_timeout_us(1000);
60             if ((c >= '0') && (c <= '9')) {
61                 putchar_raw(c);
62                 dig[n++] = c - '0';
63             }
64         }
65         printf("\n");
66         dt.month = dig[0]*10+dig[1];
67         dt.day = dig[2]*10+dig[3];
```

```
68     dt.year = dig[4]*1000+dig[5]*100+dig[6]*10+dig[7];
69     dt.dotw = 0;
70     dt.hour = dig[8]*10+dig[9];
71     dt.min = dig[10]*10+dig[11];
72     dt.sec = dig[12]*10+dig[13];
73     if (rtc_set_datetime(&dt)) {
74         break;
75     }
76 }
77
78 // Main loop: set alarm and wait
79 dt.month = -1;
80 dt.day = -1;
81 dt.year = -1;
82 dt.dotw = -1;
83 dt.hour = -1;
84 while (true) {
85     fired = false;
86     dt.min = (dt.min + 1 + (rand() % 5)) % 60;
87     rtc_set_alarm(&dt, alarm_callback);
88     printf ("Alarm set for xx:%02d:%02d\n", dt.min, dt.sec);
89     while (!fired) {
90         // do nothing
91     }
92 }
93
94 return 0;
95 }
```

GPIO, Pad and PWM

GPIO Overview

Of the 56 pins in the RP2040, 36 are capable of **General Purpose Input Output** (GPIO). These pins are grouped in two banks, the **User bank** and the **QSPI bank**. As the latter is used to connect the external Flash memory, we have the 30 pins in the user bank (GPIO00 to GPIO29) available for our use.

All 30 GPIOs can be used for digital input and output. They can also be used for other functions by attaching them to other internal peripherals:

- One of 2 PIOs (Programmable Input Output)
- One of 2 UARTs (Universal Asynchronous Receiver and Transmitter)
- One of 2 SPIs
- One of 2 I2Cs
- One of 16 PWMs (Pulse Width Modulation)
- Clock input or output
- USB VBUS management
- External interrupt requests

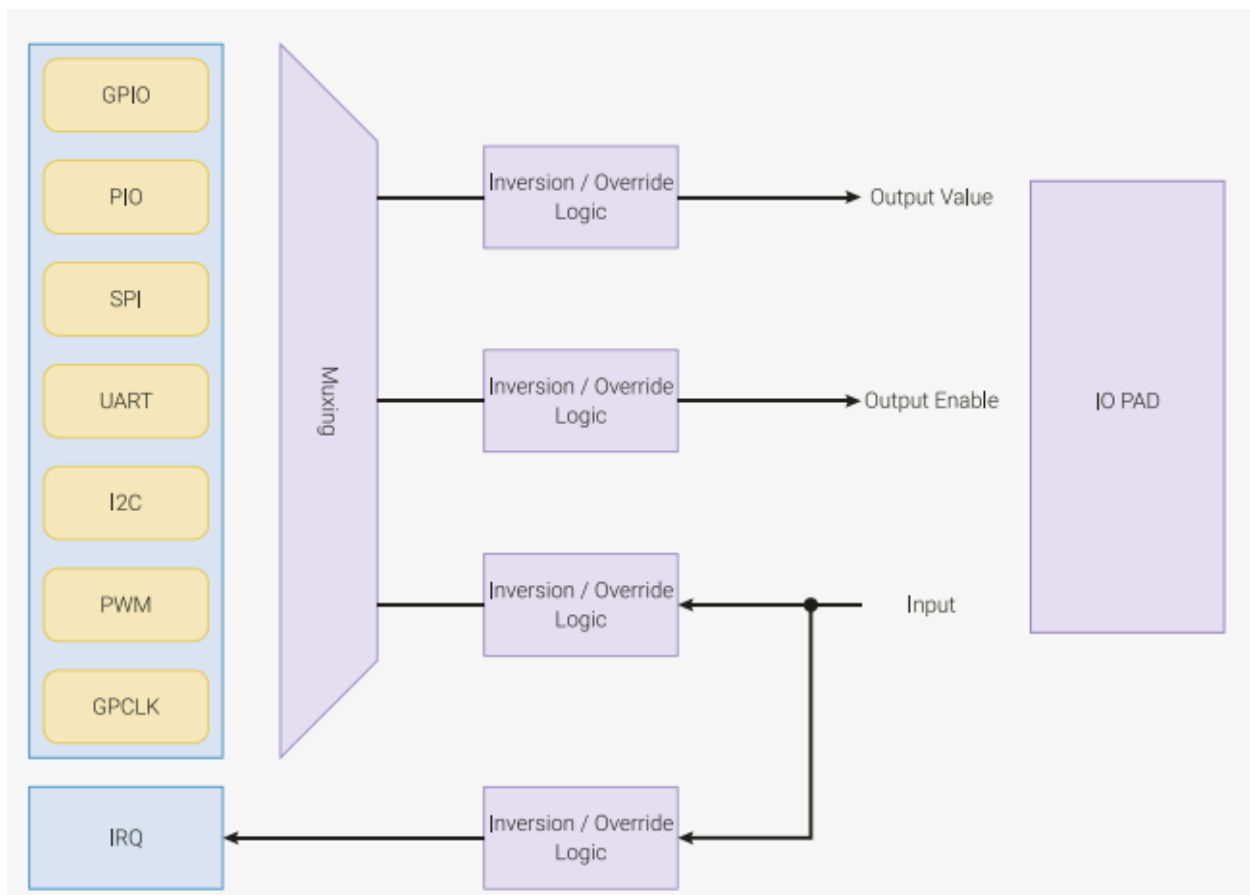
GPIO26 to GPIO29 can also be connected to the ADC (Analog to digital converter) inputs.

A GPIO register allows to select the **function of a pin**, that is, what peripheral will:

- Control the output enable (select if the pin is an output or input)
- Control the output level (used only if the output is enabled)
- Receive the pin input

GPIO registers can change (*override*) these signals, by inverting, forcing high or forcing low.

The figure below illustrates these capabilities:



GPIO Block Diagram

The three signals (output enable, output level and input) go to the *I/O PAD*. The PAD represents the electrical interface between the internal logic and the actual pin.

Function Select

Each GPIO pin has a CTRL register associated to it. This register controls the inversion or overriding of the signals and selects the function. Each pin can have up to nine functions (not counting ADC, this is not controlled here).

The following tables shows the options available for each pin.

GPIO	F1	F2	F3	F4	F5
0	SPI0 RX	UART0 TX	I2C0 SDA	PWM0 A	SIO
1	SPI0 CSn	UART0 RX	I2C0 SCL	PWM0 B	SIO
2	SPI0 SCK	UART0 CTS	I2C1 SDA	PWM1 A	SIO
3	SPI0 TX	UART0 RTS	I2C1 SCL	PWM1 B	SIO
4	SPI0 RX	UART1 TX	I2C0 SDA	PWM2 A	SIO
5	SPI0 CSn	UART1 RX	I2C0 SCL	PWM2 B	SIO
6	SPI0 SCK	UART1 CTS	I2C1 SDA	PWM3 A	SIO
7	SPI0 TX	UART1 RTS	I2C1 SCL	PWM3 B	SIO
8	SPI1 RX	UART1 TX	I2C0 SDA	PWM4 A	SIO
9	SPI1 CSn	UART1 RX	I2C0 SCL	PWM4 B	SIO
10	SPI1 SCK	UART1 CTS	I2C1 SDA	PWM5 A	SIO
11	SPI1 TX	UART1 RTS	I2C1 SCL	PWM5 B	SIO
12	SPI1 RX	UART0 TX	I2C0 SDA	PWM6 A	SIO
13	SPI1 CSn	UART0 RX	I2C0 SCL	PWM6 B	SIO
14	SPI1 SCK	UART0 CTS	I2C1 SDA	PWM7 A	SIO
15	SPI1 TX	UART0 RTS	I2C1 SCL	PWM7 B	SIO
16	SPI0 RX	UART0 TX	I2C0 SDA	PWM0 A	SIO
17	SPI0 CSn	UART0 RX	I2C0 SCL	PWM0 B	SIO
18	SPI0 SCK	UART0 CTS	I2C1 SDA	PWM1 A	SIO
19	SPI0 TX	UART0 RTS	I2C1 SCL	PWM1 B	SIO
20	SPI0 RX	UART1 TX	I2C0 SDA	PWM2 A	SIO
21	SPI0 CSn	UART1 RX	I2C0 SCL	PWM2 B	SIO
22	SPI0 SCK	UART1 CTS	I2C1 SDA	PWM3 A	SIO
23	SPI0 TX	UART1 RTS	I2C1 SCL	PWM3 B	SIO
24	SPI1 RX	UART1 TX	I2C0 SDA	PWM4 A	SIO
25	SPI1 CSn	UART1 RX	I2C0 SCL	PWM4 B	SIO
26	SPI1 SCK	UART1 CTS	I2C1 SDA	PWM5 A	SIO
27	SPI1 TX	UART1 RTS	I2C1 SCL	PWM5 B	SIO
28	SPI1 RX	UART0 TX	I2C0 SDA	PWM6 A	SIO
29	SPI1 CSn	UART0 RX	I2C0 SCL	PWM6 B	SIO

GPIO	F6	F7	F8	F9
0	PIO0	PIO1		USB OVCUR DET
1	PIO0	PIO1		USB VBUS DET
2	PIO0	PIO1		USB VBUS EN
3	PIO0	PIO1		USB OVCUR DET
4	PIO0	PIO1		USB VBUS DET
5	PIO0	PIO1		USB VBUS EN
6	PIO0	PIO1		USB OVCUR DET
7	PIO0	PIO1		USB VBUS DET
8	PIO0	PIO1		USB VBUS EN
9	PIO0	PIO1		USB OVCUR DET
10	PIO0	PIO1		USB VBUS DET
11	PIO0	PIO1		USB VBUS EN
12	PIO0	PIO1		USB OVCUR DET
13	PIO0	PIO1		USB VBUS DET

GPIO	F6	F7	F8	F9
14	PIO0	PIO1		USB VBUS EN
15	PIO0	PIO1		USB OVCUR DET
16	PIO0	PIO1		USB VBUS DET
17	PIO0	PIO1		USB VBUS EN
18	PIO0	PIO1		USB OVCUR DET
19	PIO0	PIO1		USB VBUS DET
20	PIO0	PIO1	CLOCK GPIN0	USB VBUS EN
21	PIO0	PIO1	CLOCK GPOUT0	USB OVCUR DET
22	PIO0	PIO1	CLOCK GPIN1	USB VBUS DET
23	PIO0	PIO1	CLOCK GPOUT1	USB VBUS EN
24	PIO0	PIO1	CLOCK GPOUT2	USB OVCUR DET
25	PIO0	PIO1	CLOCK GPOUT3	USB VBUS DET
26	PIO0	PIO1		USB VBUS EN
27	PIO0	PIO1		USB OVCUR DET
28	PIO0	PIO1		USB VBUS DET
29	PIO0	PIO1		USB VBUS EN

Selected SDK Functions

This functions are related to the function selection, they are in the library `hardware_gpio`:

```
void gpio_set_function (uint gpio, enum gpio_function fn)
```

Selects the function of a pin. `gpio_function` has the following options:

- `GPIO_FUNC_XIP` Flash execute in place, not used in the User bank
- `GPIO_FUNC_SPI` SPI0 or SPI1
- `GPIO_FUNC_UART` UART0 or UART1
- `GPIO_FUNC_I2C` I2C0 or I2C1
- `GPIO_FUNC_PWM` PWM
- `GPIO_FUNC_SIO` plain GPIO (digital input/output): software control via SIO (Single-Cycle IO)
- `GPIO_FUNC_PIO0`, `GPIO_FUNC_PIO1` PIO
- `GPIO_FUNC_GPCK` Clock Input or Output
- `GPIO_FUNC_USB` USB VBUS management
- `GPIO_FUNC_NULL` pin disabled

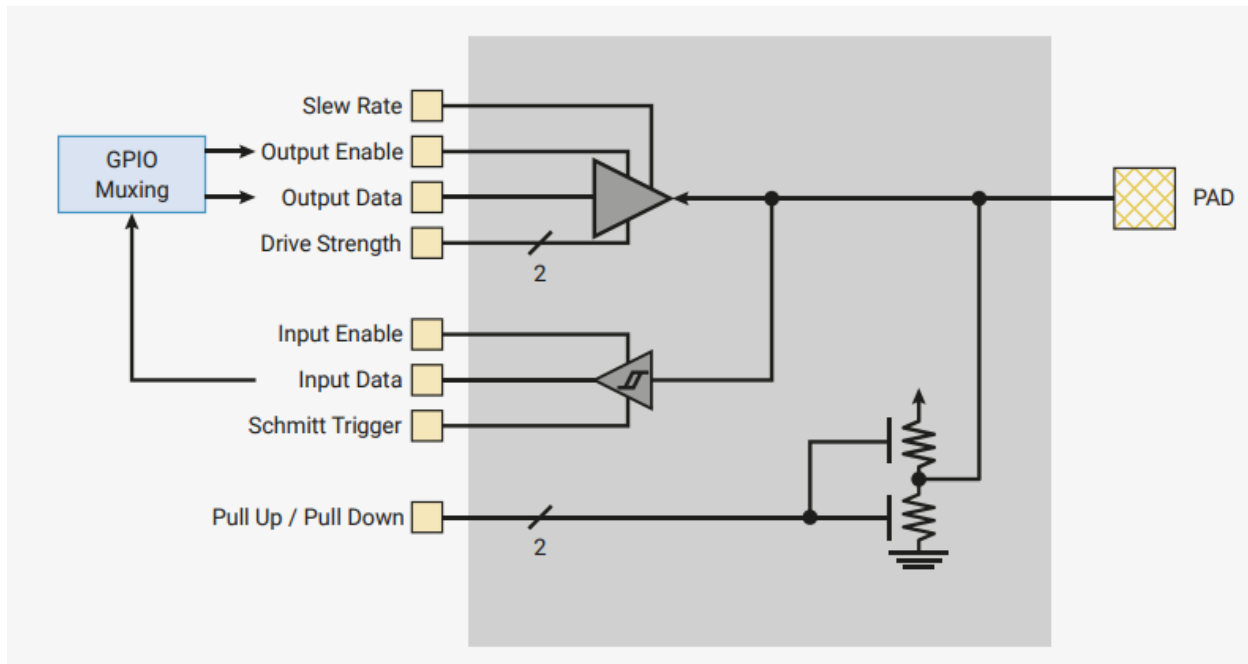
When using this function, check in the previous table what functions are available, the instance of the peripheral (for example, SPI0 or SPI1) and what peripheral signal is connected (for example, MISO, MOSI, SCK or CS for SPI).

```
enum gpio_function gpio_get_function (uint gpio)
```

Returns the current function of a pin.

PADs

Each pin has a **PAD**, an electrical interface between the internal logic and the actual pin. A logical view of it is shown bellow:

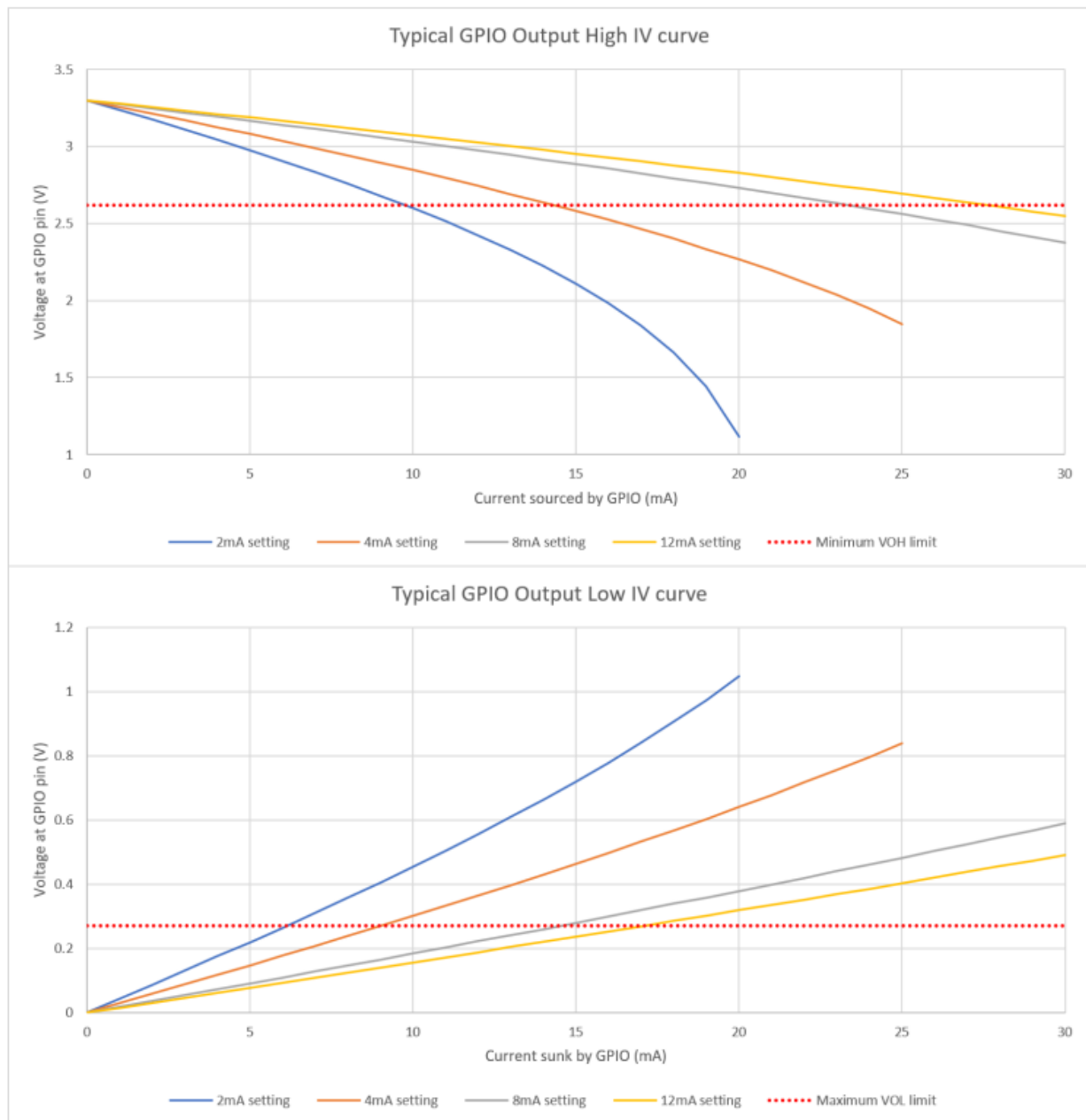


PAD Interface

The PAD has many configurations, controlled by registers:

- *Slew rate* controls how fast a pin changes state
- *Drive strength* controls how “strong” the signal is (more about that soon)
- We can enable or disable *pull-up* and *pull-down* resistors.
- We can enable or disable the *input buffer*
- We can enable or disable the *schmitt trigger* function on the input buffer. When enabled, different voltage levels are used detecting changes in the input from low to high and from high to low. This helps ignore small changes in the input signal

The graph bellow shows the effect of the drive strength configuration:



How Drive Strength Affects Output

In other words, a higher drive strength result in an output voltage closer to ideal if more current flows through the pin. Independent of this configuration, **the maximum sum of current of all GPIO pins is 50 mA**. The default drive strength is 4 mA.

While it may be tempting to set all pins to maximum strength, that is not recommended. First it is useless, as there is the 50 mA overall limit. Second if the load in a pin is capacitive, a higher strength will mean a higher current when the output is changed from low to high. When designing your application, start by summing up the currents for the less demanding pins (the one if 2mA or less).

Then compare what is left of the 50mA “budget” with what the other pins need and decide if you can use higher drive strength for them or use external transistors.

The **input buffer** has two functions: provide a high input impedance (to reduce the current into the pin and avoid to affect the voltage at the input) and to decide whether this voltage is low or high (generating the Input Data signal). The Input Enable signal can disable the buffer when we are using the pin for analog input or we are not using it and want to reduce power consumption.

The **Schmitt Trigger**, if active, introduces a hysteresis into the decision by using different decision values for low to high and high to low transitions. This reduces changes in the input value when the voltage at the pin makes small changes around the limit between low and high level.

The RP2040 provides for **pull-up** and **pull-down** resistors (with values somewhere between 50k and 80k). These resistors are useful to guarantee a known level if a pin is open. One common example is using a switch that connects a pin to ground or leave it open. By enabling the pull-up resistor we will read 1 when the switch is open and 0 when it is closed. There are many components that have outputs that work like (what is called, often not precisely, as an open collector output).

Selected SDK Functions

The C/C++ SDK has many functions to configure the PADs in the library `hardware_gpio`.

Pull-up and Pull-down control

```
static void gpio_pull_up (uint gpio)
```

Connects the pull-up resistor.

```
static void gpio_pull_down (uint gpio)
```

Connects the pull-down resistor.

```
void gpio_set_pulls (uint gpio, bool up, bool down)
```

Control both pull resistors, true means connect, false disconnect.

```
static void gpio_disable_pulls (uint gpio)
```

Disconnect both resistors.

Input buffer control

```
void gpio_set_input_enabled (uint gpio, bool enabled)
```

Enables or disables the input buffer.

```
void gpio_set_input_hysteresis_enabled (uint gpio, bool enabled)
```

Enables or disables the Schmitt Trigger function in the input buffer.

Slew-rate and Drive Strength control

```
void gpio_set_slew_rate (uint gpio, enum gpio_slew_rate slew)
```

Selects the slew rate for a GPIO pin. The options for `slew` are `GPIO_SLEW_RATE_SLOW` and `GPIO_SLEW_RATE_FAST`.

```
void gpio_set_drive_strength (uint gpio, enum gpio_drive_strength drive)
```

Sets the drive strength for a GPIO pin. The options for `drive` are `GPIO_DRIVE_STRENGTH_2MA`, `GPIO_DRIVE_STRENGTH_4MA`, `GPIO_DRIVE_STRENGTH_8MA` and `GPIO_DRIVE_STRENGTH_12MA`.

Signals Override

This functions controls *overriding* the signals. The values used are:

- `GPIO_OVERRIDE_NORMAL`: no change in the signal
- `GPIO_OVERRIDE_INVERT`: signal is inverted
- `GPIO_OVERRIDE_LOW`: signal is forced low or disabled
- `GPIO_OVERRIDE_HIGH`: signal is forced high or enabled

```
void gpio_set_irqover (uint gpio, uint value)
```

Controls the overriding of the interrupt signal.

```
void gpio_set_outover (uint gpio, uint value)
```

Controls the overriding of the output signal.

```
void gpio_set_inover (uint gpio, uint value)
```

Controls the overriding of the input signal.

```
void gpio_set_oeover (uint gpio, uint value)
```

Controls the overriding of the output enable signal.

Digital Input and Output

The **digital input and output** function corresponds to the “GPIO” block in the figure we saw at the overview. We have basically three 32 bit registers, where each bit is associated with a pin:

- `GPIO_OUT` determines the state (high or low) of the pins, if output is enabled and the pin is configured for GPIO.
- `GPIO_OE` enables or disables output, if the pin is configure for GPIO.
- `GPIO_IN` indicates the digital state (high or low) of the pin, regardless of the pin function.

There is a single set of these registers, accessible by the two ARM cores (they are part of the SIO).

Digital Output

Digital output is controlling the voltage of a pin, selecting between a low (0) and a high (1) value.

To use digital output we must:

- Select the GPIO function for the pin
- Configure the pad
- Set the initial output level
- Enable the output

These steps are normally done at initialization, as they do not need to be repeated when the level is changed.

Digital Input

The objective of **digital input** is to check if a pin has a high or low voltage level applied.

The digital input is always available, even if the output is enabled. To use digital input we must:

- Select the GPIO function for the pin
- Configure the pad

Selected SDK Functions

The SDK includes functions to make changes in multiple pins (selected by a 32-bit mask). As a single 32 bit register controls all pins, this can be done very efficiently. You probably won't have to access the registers directly; if the SDK functions do not give you the performance you need, you should use the PIO for the task.

When a mask is used, bit 0 corresponds to GPIO0, bit 1 to GPIO01 and so on until bit 29.

The following functions are in the `hardware_gpio` library.

Initialization

A pin must be initialized before other GPIO uses.

```
void gpio_init (uint gpio)
```

Initializes a pin for GPIO use. The pin is configured for input.

```
void gpio_init_mask (uint gpio_mask)
```

Initializes the pins select by `mask` for GPIO use. The pins are configured for input.

Pin Direction Control

A pin should be initialized before calling this functions.

```
static void gpio_set_dir (uint gpio, bool out)
```

Sets the direction (out = true for output, false for input) of a GPIO pin.

```
static void gpio_set_dir_all_bits (uint32_t values)
```

Sets the direction of all pins. Each bit in values correspond to a pin (0 for input, 1 for output).

```
static void gpio_set_dir_masked (uint32_t mask, uint32_t value)
```

Sets the direction of the pins selected by mask. Each bit in value correspond to a pin (0 for input, 1 for output).

```
static void gpio_set_dir_out_masked (uint32_t mask)
```

Sets the direction of the pins selected by mask to output.

```
static void gpio_set_dir_in_masked (uint32_t mask)
```

Sets the direction of the pins selected by mask to input.

Digital Input

A pin should be initialized before calling this functions.

```
static bool gpio_get (uint gpio)
```

Get current state of a pin (0 for low, 1 for high).

```
static uint32_t gpio_get_all (void)
```

Get current state of all pins. Each bit in the result corresponds to a pin (0 for low, 1 for high).

Digital Output

A pin should be initialized and configure for output before calling this functions.

```
static void gpio_put (uint gpio, bool value)
```

Changes the state (value = true for high, false for low) of a GPIO pin.

```
static void gpio_put_all (uint32_t value)
```

Changes the state of all pins. Each bit in value correspond to a pin (0 for low, 1 for high).

```
static void gpio_put_masked (uint32_t mask, uint32_t value)
```

Changes the state of the pins selected by mask. Each bit in value correspond to a pin (0 for low, 1 for high).

```
static void gpio_set_mask (uint32_t mask)
```

Sets (output high level) the GPIO pins selected by mask.

```
static void gpio_clr_mask (uint32_t mask)
```

Clear (output low level) the GPIO pins selected by mask.

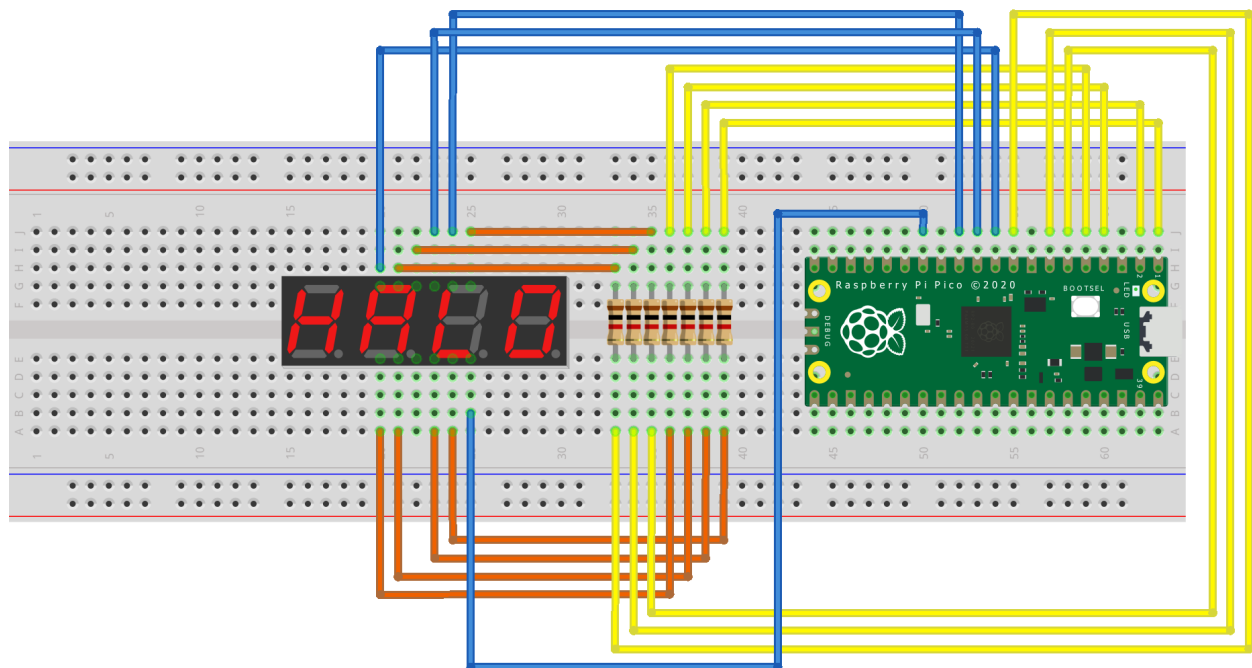
```
static void gpio_xor_mask (uint32_t mask)
```

Invert the state of the GPIO pins selected by mask

Examples

Digital Output Example

In this example we are going to drive a four digit seven segment common cathode display to continuously count from 0000 to 9999.



fritzing

Seven Segment Display Connection

Segments and the common cathodes are connected to GPIO pins. 1K resistors in each segment limit its current to 1.4 mA (for the particular display used). The common cathode will supply up to the sum of these currents ($7 \times 1.4 = 9.8$ mA), so we need to configure a greater drive strength than the default.

By using the `gpio_put_masked` function we can change all segments in a single call.

Digital Output Example

```

1  /**
2   * @file gpio7segment.c
3   * @author Daniel Quadros
4   * @brief Example of using the GPIO in the RP2040 to drive a
5   *        4 digit 7 segment common anode display
6   * @version 0.1
7   * @date 2022-07-12
8   *
9   * @copyright Copyright (c) 2022, Daniel Quadros
10  *
11  */
12
13  #include <stdio.h>
14  #include "pico/stdlib.h"
15  #include "hardware/gpio.h"
16  #include "hardware/sync.h"
17
18  // Display connections
19  // Segments:  A:6 B:4 C:1 D:2 E:3 F:5 G:0
20  // Digits:    1:7 2:8 3:9 4:10
21  #define SEGMENTS_MASK    0x0007F
22  #define DIGITS_MASK      0x00780
23  #define DIGIT_1          7
24  #define DIGIT_2          8
25  #define DIGIT_3          9
26  #define DIGIT_4         10
27
28  // Digit selection GPIOs
29  int digit[] = { DIGIT_1, DIGIT_2, DIGIT_3, DIGIT_4 };
30
31  // What segments to turn on for each digit
32  int segments[] = { // AFB EDCG  0 = on, 1 = off
33      0x01,          // 000 0001    --A--
34      0x6D,          // 110 1101    F   B
35      0x22,          // 010 0010    --G--
36      0x28,          // 010 1000    E   C
37      0x4C,          // 100 1100    --D--
38      0x18,          // 001 1000
39      0x10,          // 001 0000
40      0x2D,          // 010 1101
41      0x00,          // 000 0000
42      0x08           // 000 1000

```

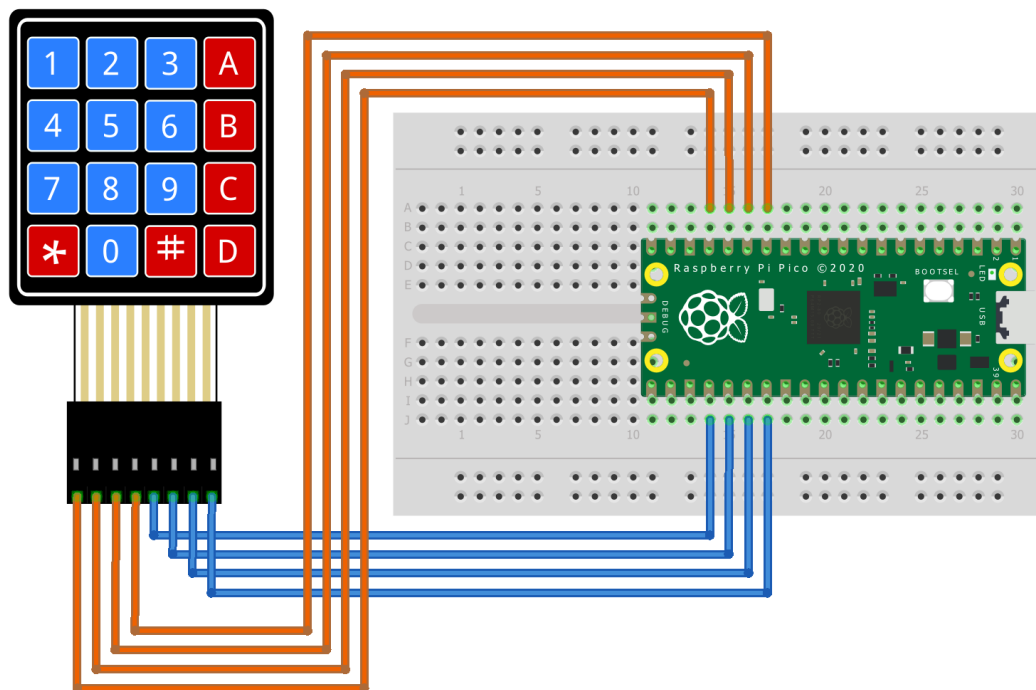
```
43 };
44
45 // Timer to update the display
46 struct repeating_timer timer;
47
48 // Value to show on display
49 volatile int value[4];
50
51 // Local routines
52 static void init(void);
53 static bool updateDisplay(struct repeating_timer *t);
54
55 // Main Program
56 int main() {
57     init();
58     while (1) {
59         // Increment value
60         int i = 3;
61         while ((i >= 0) && (value[i]==9)) {
62             value[i] = 0;
63             i--;
64         }
65         if (i >= 0) {
66             value[i]++;
67         }
68         // Wait 1 second
69         sleep_ms(1000);
70     }
71     return 0;
72 }
73
74 // Initialization
75 void init() {
76     int i;
77
78     // GPIO init
79     gpio_init_mask (SEGMENTS_MASK | DIGITS_MASK);
80     gpio_set_dir_masked (SEGMENTS_MASK | DIGITS_MASK, SEGMENTS_MASK | DIGITS_MASK);
81     for (i = 0; i < 4; i++) {
82         gpio_set_drive_strength (digit[i], GPIO_DRIVE_STRENGTH_12MA);
83     }
84
85     // Update a digit every 5 milliseconds
```

```
86     add_repeating_timer_ms(5, updateDisplay, NULL, &timer);
87 }
88
89 // Update Display
90 bool updateDisplay(struct repeating_timer *t) {
91     static int nDig = 3;
92
93     gpio_put (digit[nDig], false); // turn off previous digit
94     nDig = (nDig + 1) & 3; // moves on to next digit
95
96     // set up segments
97     gpio_put_masked (SEGMENTS_MASK, segments[value[nDig]]);
98
99     gpio_put (digit[nDig], true); // turns on current digit
100
101     return true; // keep calling this routine
102 }
```

Digital Input Example

Here we are going to interface a 4x4 Matrix Keypad. This keypad has sixteen keys connected in a 4 row by 4 column matrix.

We will connect the 4 rows to GPIOs configured for output and the 4 columns to GPIOs configured for input with pull-down resistor enabled.



fritzing

Keypad Connection

To detect the keys pressed, we will put a HIGH level in one row at a time and read the level at the columns. A pressed key will read as HIGH and a released key will read as LOW.

The detected keys will be sent through stdio.

Digital Input Example

```

1  /**
2   * @file gpio7segment.c
3   * @author Daniel Quadros
4   * @brief Example of using the GPIO in the RP2040 to
5   *        read a 4x4 matrix keypad
6   * @version 0.1
7   * @date 2022-07-14
8   *
9   * @copyright Copyright (c) 2022, Daniel Quadros
10  *
11  */
12
13 #include <stdio.h>
14 #include "pico/stdlib.h"
15 #include "hardware/gpio.h"
16 #include "hardware/sync.h"

```

```

17
18 // GPIOs
19 // Rows:    GPIO10 to GPIO13
20 // Columns: GPIO18 to GPIO21
21 #define nRows    4
22 #define nColumns 4
23 static const int firstRow = 10;
24 static const int firstColumn = 18;
25
26 // GPIO masks
27 static uint32_t rowMask;
28 static uint32_t columnMask;
29
30 // Timer to scan the keypad
31 static struct repeating_timer timer;
32
33 // Columns readings
34 static const int DEBOUNCE = 5;
35 static uint32_t kp_debounced[nRows];
36 static uint32_t kp_debouncing[nRows];
37 static int debounceCounter[nRows];
38
39 // Keys queue
40 #define sizeQueue 5
41 static int inQueue = 0, outQueue = 0;
42 static char queue[sizeQueue+1];
43
44 // Keypad decoding
45 static int decod[nRows][nColumns] = {
46     { '1', '2', '3', 'A' },
47     { '*', '0', '#', 'D' },
48     { '7', '8', '9', 'C' },
49     { '4', '5', '6', 'B' }
50 };
51
52 // Local routines
53 static uint32_t buildMask(int first, int n);
54 static void init(void);
55 static bool scanKeypad(struct repeating_timer *t);
56 static int readKey(void);
57
58 // Main Program
59 int main() {

```

```
60     // Init stdio
61     stdio_init_all();
62     while (!stdio_usb_connected()) {
63         sleep_ms(100);
64     }
65
66     printf("\nKeypad GPIO Input Example\n");
67     init();
68     while (1) {
69         int key = readKey();
70         if (key != -1) {
71             printf ("Key = %c\n", key);
72         }
73         sleep_ms(1);
74     }
75     return 0;
76 }
77
78 // Utility routine to build a mask for 'n' pins starting from 'first'
79 static uint32_t buildMask(int first, int n) {
80     uint32_t mask = 0;
81     for (int i = 0; i < n; i++) {
82         mask |= 1 << (first+i);
83     }
84     return mask;
85 }
86
87 // Initialization
88 static void init() {
89
90     // Build masks
91     rowMask = buildMask (firstRow, nRows);
92     columnMask = buildMask (firstColumn, nColumns);
93
94     // GPIO init
95     gpio_init_mask (rowMask | columnMask);
96     gpio_set_dir_masked (rowMask | columnMask, rowMask);
97     for (int i = 0; i < nColumns; i++) {
98         gpio_pull_down(firstColumn+i);
99     }
100
101     // Scan keypad every 10 milliseconds
102     add_repeating_timer_ms(10, scanKeypad, NULL, &timer);
```

```

103 }
104
105 // Scan the current row of the keypad
106 static bool scanKeypad(struct repeating_timer *t) {
107     static int curRow = firstRow;
108     static int countRow = 0;
109
110     // Turn on current row
111     gpio_put_masked (rowMask, 1 << curRow);
112
113     // Read columns
114     uint32_t current = gpio_get_all() & columnMask;
115
116     // Debounce
117     if (current != kp_debouncing[countRow]) {
118         // reading changed, start debouncing again
119         kp_debouncing[countRow] = current;
120         debounceCounter[countRow] = 0;
121     } else if (debounceCounter[countRow] <= DEBOUNCE) {
122         if (debounceCounter[countRow] == DEBOUNCE) {
123             // consider value stable
124             if (kp_debounced[countRow] != current) {
125                 // Find key pressed
126                 uint32_t dif = kp_debounced[countRow] ^ current;
127                 int i = 0;
128                 while (i < nColumns) {
129                     uint32_t mask = 1 << (i+firstColumn);
130                     if (((dif & mask) != 0) && ((current & mask) != 0)) {
131                         // there is a change and key is pressed
132                         break;
133                     }
134                     i++;
135                 }
136                 if (i < nColumns) {
137                     int key = decod[countRow][i];
138                     int aux = inQueue+1;
139                     if (aux > sizeQueue) {
140                         aux = 0;
141                     }
142                     if (aux != outQueue) {
143                         queue[inQueue] = key;
144                         inQueue = aux;
145                     } else {

```

```

146             // queue is full, ignore key
147         }
148     }
149     // uodate debounced status
150     kp_debounced[countRow] = current;
151 }
152 }
153     debounceCounter[countRow]++;
154 }
155
156 // Move on to next row
157 if (++countRow == nRows) {
158     countRow = 0;
159     curRow = firstRow;
160 } else {
161     curRow++;
162 }
163
164 return true; // keep executing
165 }
166
167
168 // Read a key from the key queue, returns -1 if queue empty
169 static int readKey(void) {
170     int key = -1;
171     if (inQueue != outQueue) {
172         key = queue[outQueue];
173         if (outQueue++ == sizeQueue) {
174             outQueue = 0;
175         }
176     }
177     return key;
178 }

```

The main action in this example is in `scanKeypad()`. This routine is called every 10 milliseconds and reads one row each time. A two stage processing is done to discard short changes (*debounce*). First the current reading is checked against the value that is been validated (`kp_debouncing`). Only after reading the same value multiple times it is checked against the previous validated value (`kp_debounced`). Detected keys are put in a queue to be read by the main loop and sent through `stdio`.

GPIO Interrupts

A digital signal can cause an interrupt through a GPIO pin on four events:

- Level high: the signal is at high level (1)
- Level low: the signal is at low level (0)
- Edge rise: the signal changes from low to high level (0 to 1)
- Edge fall: the signal changes from high to low level (1 to 0)

There are three possible destinations for GPIO interrupts: processor 0, processor 1 and dormant wake (we will learn about this latter in this chapter).

GPIO interrupts are ORed per bank and destination. In the NVIC of each processor there are two sources related to GPIO interrupts, one for BANK0 and one for BANK1 (QSPI). It is up to the handler of the interrupt to figure out what GPIO pin triggered the interrupt.

Level Interrupts

Level interrupts have no memory: the interrupt is active as long as the signal has the selected level and inactive as soon as the pin changes to the other level.

In a typical use, the interrupt signal is generated by some device indicating that it needs attention.

In a simple scenario, there is only one cause for the interrupt. The handler will do some interaction with the device and it will change the signal. If the device does not change the signal during the handling of the interrupt, it will have to be masked and re-enabled latter.

In a more complicated scenario, there are multiple causes for the interrupt and the interrupt will keep firing until all causes have been treated. The handler may try to treat all causes in a single interrupt or treat just one and let a new interrupt occur if there are others.

If the signal changes before the interrupt is treated the interrupt will be lost. In this case edge interrupt may be a better option.

Edge Interrupts

Edge interrupts are saved in the INTR register. A write in the corresponding bit in the INTR register clears the interrupt.

Again, one typical use is when the interrupt signal is generated by some device indicating that it needs attention. The handler will attend the device; at some future point the signal will change to the other level. Only when it changes a second time will a new interrupt be generated.

Things can get complicated if there are multiple causes for the interrupt or if a new interrupt needs to be generated while the interrupt is disabled or been treated, in these cases level interrupt may be a better option.

Another typical use for edge interrupts is when the signal is generated by a sensor. In this case, we use interrupts to detect changes in the sensor output.

Selected SDK Functions

These functions are in the library `hardware_gpio` and affect only the processor core that is calling it.

The SDKs implements two kinds of routines that are called in response to a GPIO event: a “normal” callback and a “raw” callback. There is only one “normal” callback for each processor, but there can be multiple “raw” callbacks.

When `gpio_set_irq_callback()` is called with a non-null callback, `gpio_default_irq_handler()` (implemented in `hardware_gpio/gpio.c`) is added as a shared handler for `IO_IRQ_BANK0`. This routine will check events on all the pins and, for each event set, acknowledge it and call the registered “normal” callback (unless a “raw” callback was registered). The “normal” callback receives the pin number and event mask as parameters.

A *raw callback*, registered via the `add_raw_irq_handler` functions, is added as a shared handler for `IO_IRQ_BANK0`. This callback has no parameters.

This means that when a `IO_IRQ_BANK0` is triggered:

- The default SDK interrupt handler is activated and calls the shared handlers registered. That includes the raw callbacks and the default GPIO irq handler.
- The default GPIO irq handler will call the “normal” callback for pins that do not have a raw callback registered.

```
void gpio_set_irq_enabled (uint gpio, uint32_t event_mask, bool enabled)
```

Enables (enable = 1) or disable (enable = 0) interrupts on the current processor for pin `gpio` on the events select by `event_mask`:

- `GPIO_IRQ_LEVEL_LOW`
- `GPIO_IRQ_LEVEL_HIGH`
- `GPIO_IRQ_EDGE_FALL`
- `GPIO_IRQ_EDGE_RISE`

You can select multiple events by ORing these constants.

You should set a callback before enabling interrupts.

This function does not enable or disable `IO_IRQ_BANK0`, you must use `irq_set_enabled` for that.

```
void gpio_set_irq_callback (gpio_irq_callback_t callback)
```

This function changes the “normal” callback for `gpio` interrupts in the current processor. The callback must be a void function with two parameters: the pin number and a mask of the pending events.

If callback is null and there was a previous callback, `gpio_default_irq_handler()` is unregistered.

If callback is non-null and there was no previous callback, `gpio_default_irq_handler()` is registered as a shared handler for `IO_IRQ_BANK0`.

```
void gpio_set_irq_enabled_with_callback (uint gpio, uint32_t event_mask, bool enabled,
gpio_irq_callback_t callback)
```

This routine combines the previous two functions. If `enabled` is true, also enables `IO_IRQ_BANK0`.

Notice that the callback will be used for events in all pins that do not have a raw callback associated (not just `gpio`).

```
static uint32_t gpio_get_irq_event_mask (uint gpio)
```

Returns a mask that indicates the pending events for pin `gpio`. The mask is an OR of `GPIO_IRQ_LEVEL_LOW`, `GPIO_IRQ_LEVEL_HIGH`, `GPIO_IRQ_EDGE_FALL` and `GPIO_IRQ_EDGE_RISE`.

```
void gpio_acknowledge_irq (uint gpio, uint32_t event_mask)
```

Acknowledge (clears) the events indicated by `event_mask` for pin `gpio`. The mask is an OR of `GPIO_IRQ_LEVEL_LOW`, `GPIO_IRQ_LEVEL_HIGH`, `GPIO_IRQ_EDGE_FALL` and `GPIO_IRQ_EDGE_RISE`.

```
void gpio_add_raw_irq_handler_with_order_priority_masked (uint gpio_mask, irq_handler_t
handler, uint8_t order_priority)
```

Registers a raw callback for multiple GPIO pins (defined by `gpio_mask`) with priority `order_priority`. The handler must be a void function with no parameters.

```
static void gpio_add_raw_irq_handler_with_order_priority (uint gpio, irq_handler_t
handler, uint8_t order_priority)
```

Registers a raw callback for GPIO pin `gpio` with priority `order_priority`. The handler must be a void function with no parameters.

```
void gpio_add_raw_irq_handler_masked (uint gpio_mask, irq_handler_t handler)
```

Registers a raw callback for multiple GPIO pins (defined by `gpio_mask`) with default priority. The handler must be a void function with no parameters.

```
static void gpio_add_raw_irq_handler (uint gpio, irq_handler_t handler)
```

Registers a raw callback for GPIO pin `gpio` with default priority. The handler must be a void function with no parameters.

```
void gpio_remove_raw_irq_handler_masked (uint gpio_mask, irq_handler_t handler)
```

Unregisters the raw callbacks for multiple GPIO pins (defined by `gpio_mask`).

```
static void gpio_remove_raw_irq_handler (uint gpio, irq_handler_t handler)
```

Unregisters the raw callback for GPIO pin `gpio`.

Example

This example will register the EDGE events for a pin. To test it, connect a button between GPIO16 and GND and look at messages sent to stdio.

Depending on the button and the way you press it, you may see that a single press or release will generate multiple events (even a RISE and FALL in the same interrupt). This occurs before buttons are not perfect, they may close or open rapidly multiple times before stabilizing. This is called *bouncing* (but not always caused by the bounce of a contact up and down).

The example enables the Schmitt trigger to reduce bouncing a little. You can also experiment connecting a 0.1 uF capacitor in parallel to the button (this is a very crude example of *hardware debouncing*). A good guide to debouncing can be found at <http://www.ganssle.com/debouncing.htm>.

GPIO Interrupt Example

```

1  /**
2   * @file gpiointerrupt.c
3   * @author Daniel Quadros
4   * @brief Example of using GPIO interrupts in the RP2040
5   * @version 0.1
6   * @date 2022-10-19
7   *
8   * @copyright Copyright (c) 2022, Daniel Quadros
9   *
10  */
11
12  #include <stdio.h>
13  #include <time.h>
14
15  #include "pico/stdlib.h"
16  #include "hardware/gpio.h"
17
18  #define millis() to_ms_since_boot(get_absolute_time())
19
20  // A button is connected between this pin and ground
21  #define BUTTON_PIN 16
22
23  // Structure to represent a GPIO event
24  typedef struct {
25      uint32_t event_mask;
26      uint32_t event_time;
27  } GPIO_EVENT;
28
29  // GPIO event queue

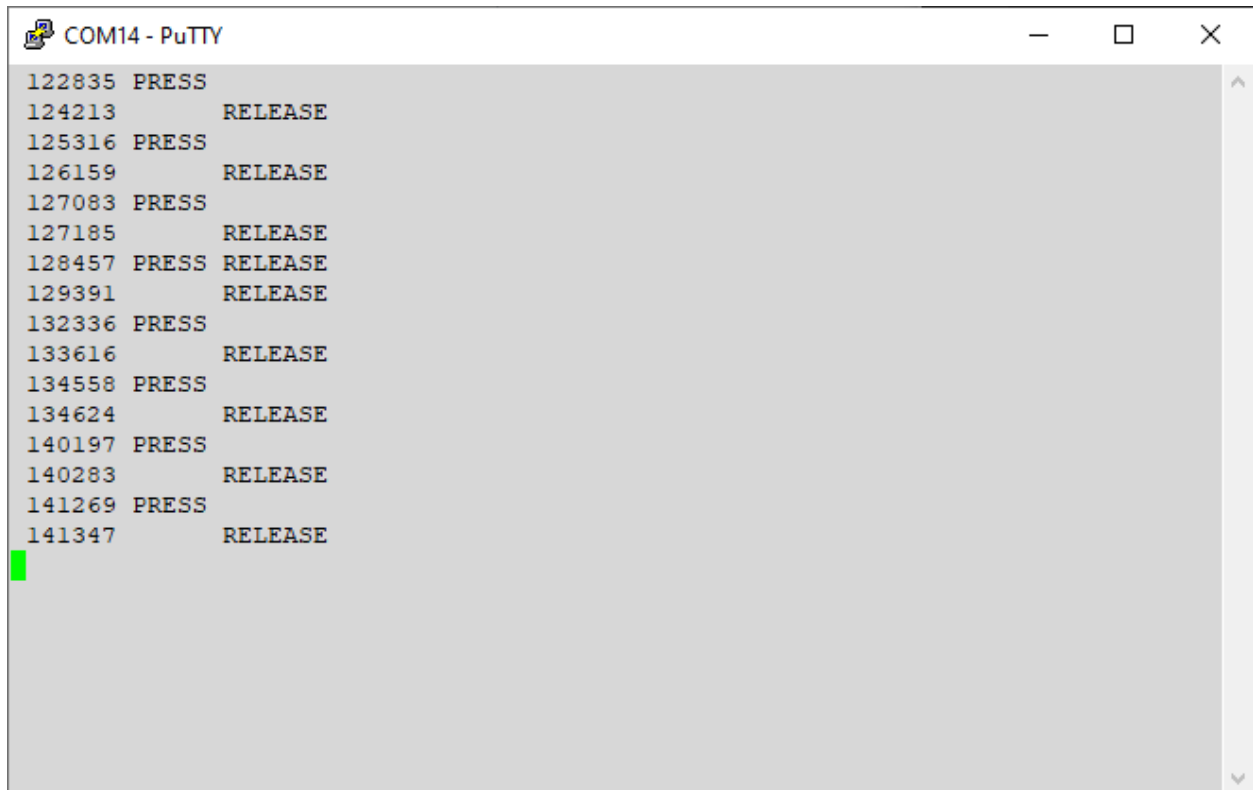
```

```

30 #define MAX_EVENTS 100
31 static GPIO_EVENT event_queue[MAX_EVENTS+1];
32 static volatile int event_in = 0; // where to place next event
33 static int event_out = 0; // where to take out next event
34
35
36 // Interrupt handler
37 void gpio_interrupt (uint gpio, uint32_t events) {
38     // set up the event information
39     event_queue[event_in].event_mask = events;
40     event_queue[event_in].event_time = millis();
41     // check if there is space for it in the queue
42     int aux = event_in + 1;
43     if (aux > MAX_EVENTS) {
44         aux = 0;
45     }
46     if (aux != event_out) {
47         // Ok, advance the input index
48         event_in = aux;
49     }
50 }
51
52
53 // Main Program
54 int main() {
55
56     // Init stdio0
57     stdio_init_all();
58
59     // Init the button pin
60     gpio_init(BUTTON_PIN);
61     gpio_set_dir(BUTTON_PIN, GPIO_IN);
62     gpio_pull_up(BUTTON_PIN);
63     gpio_set_input_hysteresis_enabled(BUTTON_PIN, true);
64
65     // Attach our callback to the gpio interrupt
66     gpio_set_irq_callback (gpio_interrupt);
67     gpio_set_irq_enabled(BUTTON_PIN, GPIO_IRQ_EDGE_FALL | GPIO_IRQ_EDGE_RISE, true\
68 );
69     irq_set_enabled(IO_IRQ_BANK0, true);
70
71     // main loop
72     while (1) {

```

```
73         // Print out recorded events
74         while (event_in != event_out) {
75             // print an event
76             printf ("%7d %s %s\n", event_queue[event_out].event_time,
77                 (event_queue[event_out].event_mask & GPIO_IRQ_EDGE_FALL) ?
78                 "PRESS" : " ",
79                 (event_queue[event_out].event_mask & GPIO_IRQ_EDGE_RISE) ?
80                 "RELEASE" : " ");
81         };
82         // remove it from the queue
83         int aux = event_out;
84         if (++aux > MAX_EVENTS) {
85             aux = 0;
86         }
87         event_out = aux;
88     }
89     sleep_ms(10);
90 }
91 return 0;
92 }
```



```
COM14 - PuTTY
122835 PRESS
124213     RELEASE
125316 PRESS
126159     RELEASE
127083 PRESS
127185     RELEASE
128457 PRESS RELEASE
129391     RELEASE
132336 PRESS
133616     RELEASE
134558 PRESS
134624     RELEASE
140197 PRESS
140283     RELEASE
141269 PRESS
141347     RELEASE
```

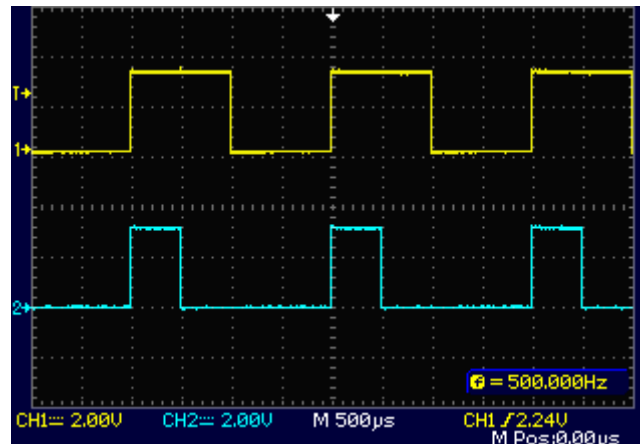
Example of GPIO events

PWM

Pulse width modulation (PWM) is a technique where we have pulses sent in regular intervals (the *frequency*) and control the time the pulse stays high. The ratio between the high time and the full pulse time is the *duty cycle*.

For example, suppose we have a signal that stays high for 1 millisecond and low for 1 millisecond. The frequency is 500Hz (we have one cycle each 2 milliseconds or 500 cycles per second). The duty cycle is 50% (what we call a *square wave*).

If instead the signal stays high for 0.5 milliseconds and low for 1.5 milliseconds, the frequency is the same 500Hz, but the duty cycle is now 25%.



PWM Example Waveforms

There are multiple uses for PWM. One example is controlling servo motors (where the duty cycle determines the position of the motor shaft). Another is to generate something alike a analog signal, as the average level changes as we change the duty cycle (we can see this by using PWM to control the brightness of an LED).

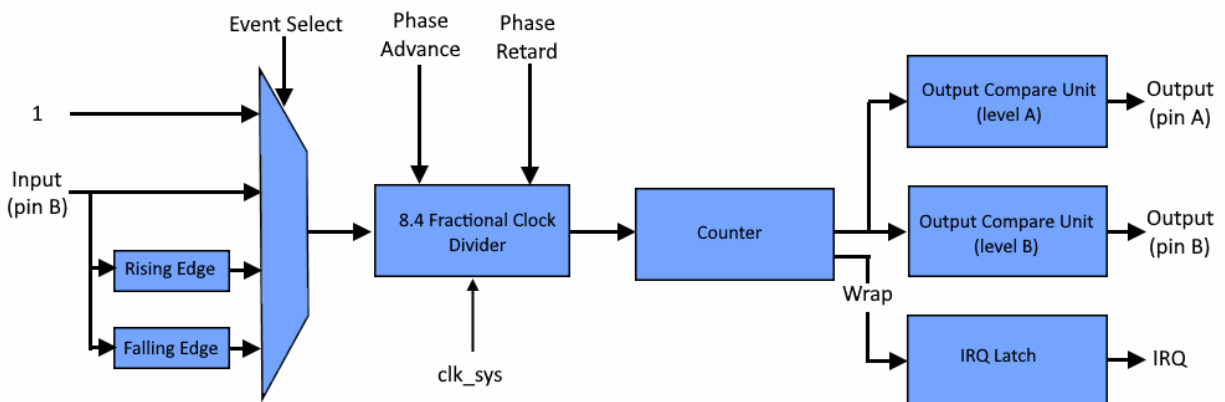
The RP2040 has dedicated hardware to generate PWM and measure frequency and duty cycle of pulses. There are two *PWM blocks*, each with eight *slices*.

PWM Slice

A PWM slice is the basic unit for generating PWM in the RP2040. There are 16 slices available. Each slice has:

- A 16 bit counter
- A 8.4 fractional clock divider
- Two independent output channels, each with a duty cycle that can go from 0% to 100%
- Dual slope and trailing edge modulation
- Edge-sensitive input mode for frequency measurement
- Level-sensitive input mode for duty cycle measurement
- Configurable counter wrap value
- Interrupt request and DMA request on counter wrap

The figure bellow show a logical view of a slice.



PWM Slice

Before going into the details, let's understand the basic operation:

- The counting is enabled
 - continuously all the time (PWM generation),
 - continuously while the level of the input pin is high (duty cycle measurement).
 - once on an edge in the input pin (frequency measurement),
- The clock divider reduces the rate of enable pulses, controlling the advance of the counter.
- When the counter reaches its wrap value it goes back to zero.
- If we are generating PWM, the high level will end when the counter reaches a certain count (*compare level*). There are two levels per slice, so we can control independently the duty cycle for two pins, named “A” and “B” (but they will have the same frequency).
- If we are measuring a signal, the counter value when it stops will give us the measurement.

Pins Assignment

All 30 GPIO pins can be used for PWM, but:

- As there are only 16 slices you can generate at most 16 different signals (if you connect the same slice output to two GPIOs they will output the same thing).
- Each slice can do only one measurement, with input in the “B” pin. So you have at most 8 measurements, and the “A” pin of the slice should not be used as PWM output (as the counting will be controlled by the input). If the “B” pin is used for input and you connect it to two GPIOs, an OR will be done between the two GPIOs.
- If you are generating PWM, the two outputs of a slice will have the same frequency.

GPIO	0	1	2	3	4	5	6	7	8	9
Channel	0A	0B	1A	1B	2A	2B	3A	3B	4A	4B

GPIO	10	11	12	13	14	15	16	17	18	19
Channel	5A	5B	6A	6B	7A	7B	0A	0B	1A	1B

GPIO	20	21	22	23	24	25	26	27	28	29
Channel	2A	2B	3A	3B	4A	4B	5A	5B	6A	6B

Clock Divider for PWM Generation

Each slice can have a different clock, created by dividing the `clk_sys` (125 MHz, if you haven't changed it) by a *fractional divider*. This divider has 12 bits, 8 for the integer part and 4 for the fractional part (in units of 1/16).

The maximum division available is 256 (obtained by setting the the divisor to zero), resulting in a clock of about 488 kHz. The frequency of the generated signal is (in the simple case) the division of this clock by the 16 bit wrap value (plus one, as the counter goes from 0 to the wrap value), so we can go down to about 7.5 Hz. If you need to generate lower frequency signals, you can use the system timer interrupt or the PIO.

When selecting the divisor, notice that the higher the clock the more precision you can get in the duty cycle. For a simple example let's suppose we want to use a frequency of 10 kHz:

- If we choose a divisor of 250, the clock will be 500Khz. To get the 10 kHz we need to wrap the counter at 49. This gives us only 50 options (0 to 49) for the wrap value (and duty cycle).
- If we choose a divisor of 12.5, the clock will be 10 MHz. We wrap the counter at 999 to get 10 kHz and get 1000 options for the duty cycle.

Basic PWM Generation

To generate a PWM signal, the counter will be in *free-running* mode, where it will always be counting. This is the default mode and in it both A and B pins will be output.

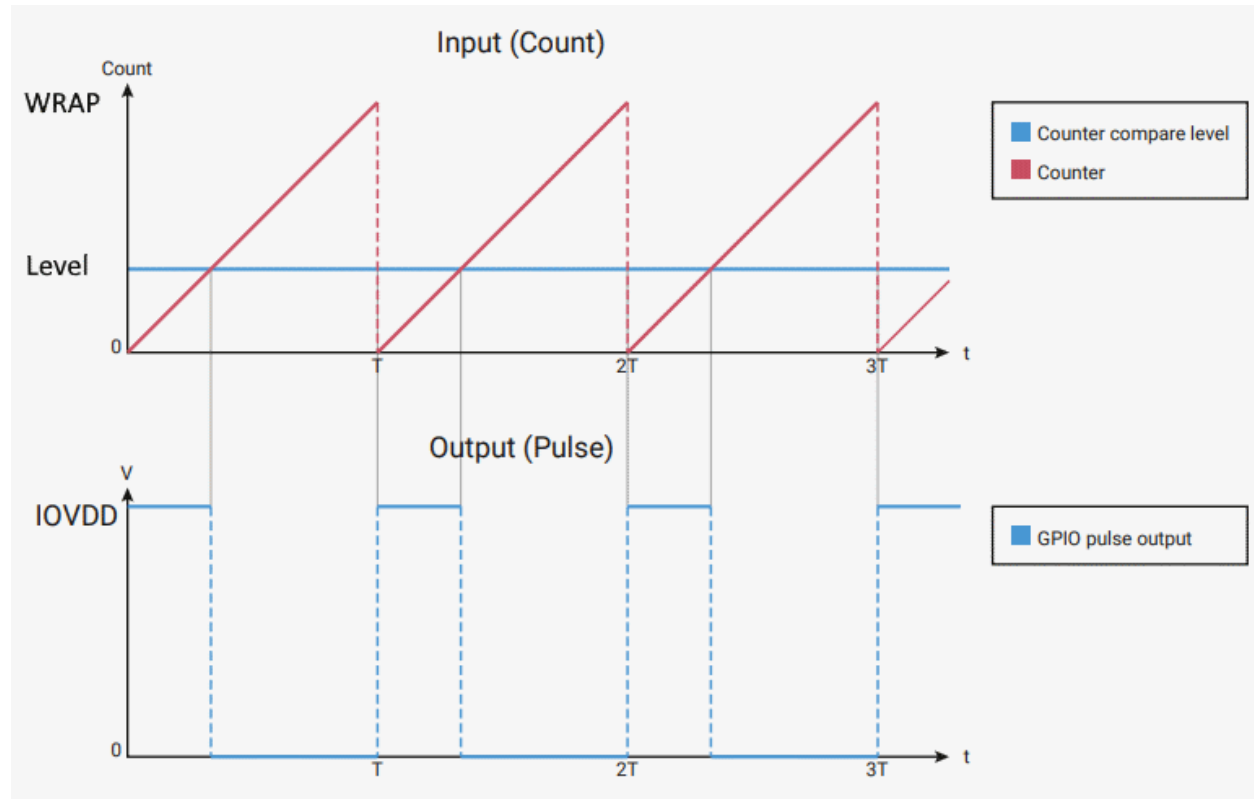
Let's look at the steps for generating a PWM signal:

- We select the pin we will use and look up to what slice and PWM pin it can be connected.
- We select the clock that will be used.
- We calculate the wrap value, based in the selected clock and the desired frequency. In most applications we will not change the frequency during operation.
- We set the initial duty cycle
- We set the function of the pin to PWM
- We configure the PAD for output
- When needed, we change the duty cycle by changing the compare level

In this basic mode, the counter will count from zero to the wrap value (inclusive), so the frequency of the signal will be

$$f_{\text{sys}} / (\text{clock divisor} * (\text{wrap value} + 1))$$

The output signal will be in the high value until the counter equals the compare level:



Basic PWM Generation

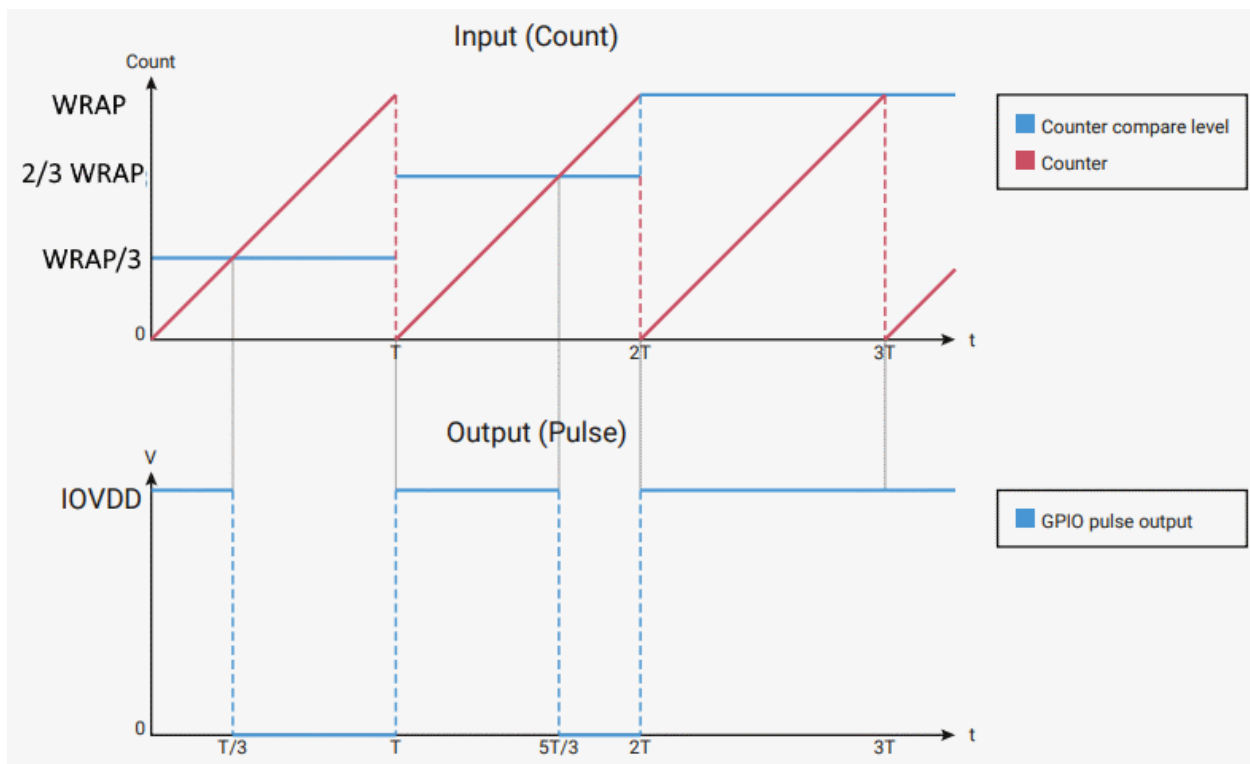
The configuration of the PWM slice is done through the following registers (that should be accessed through the SDK functions):

- DIV: clock divisor
- TOP: wrap value
- CC: counter compare (this register holds two 16 bit values, one for each output)

Some Fine Details of PWM Generation

The RP2040 supports 0% and 100% duty cycle with no glitch (the output signal will be always low or high). This is done by setting the compare level to 0 and to (wrap value + 1).

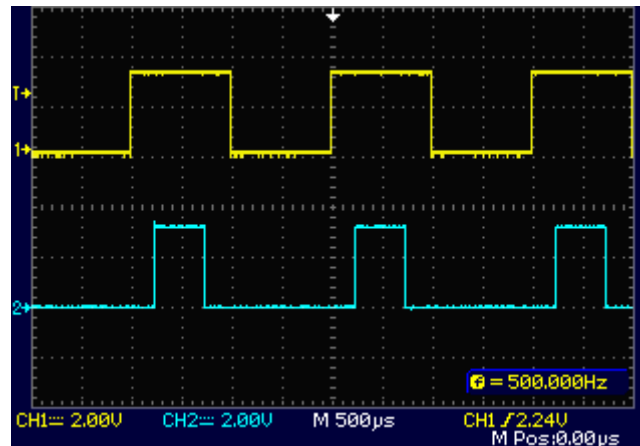
When you change the TOP or CC register, the new value will be applied when the counter wraps to zero. This is important to guarantee that the change will not occur at such a time that we get a pulse too short or too long.



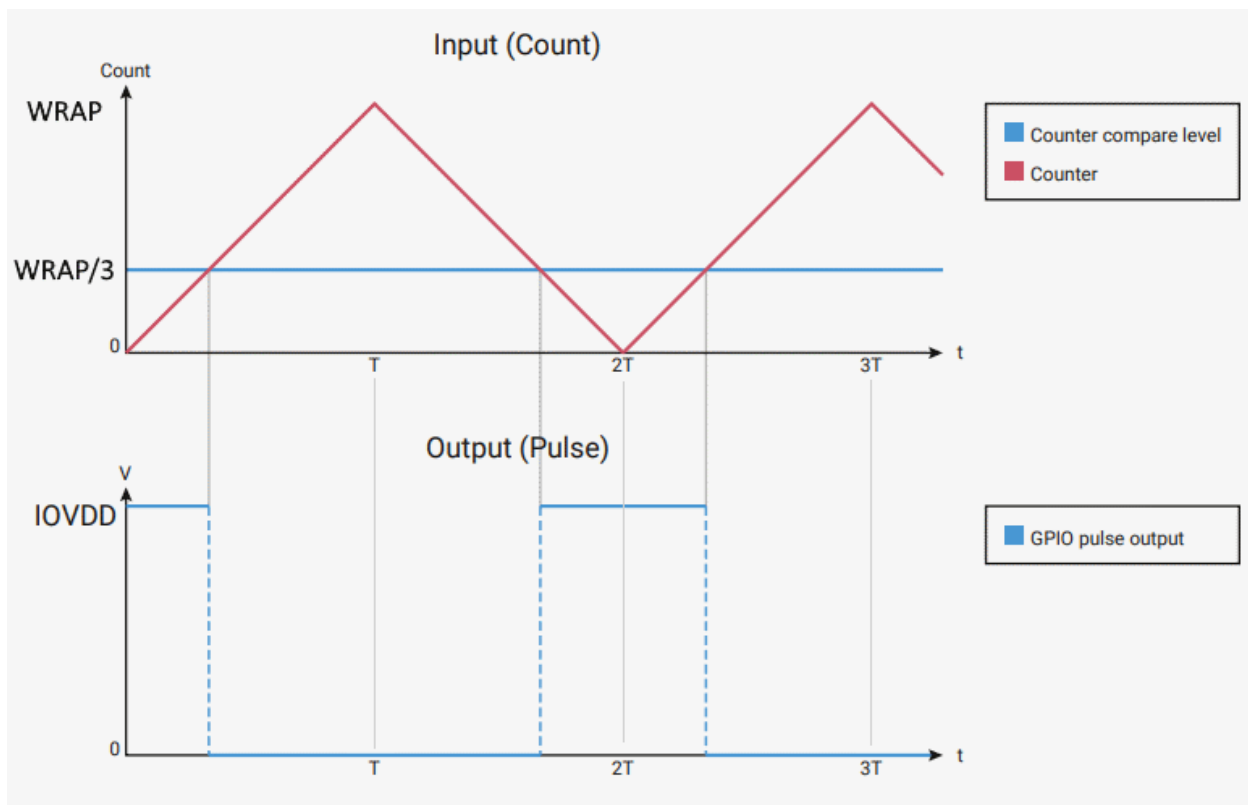
Changing Compare Level with PWM running

Sometimes we need to generate two (or more) synchronized PWM signals. There is a global PWM enable register that allows to start (or stop) multiple slices at the same time. If both use the same frequency, the pulse will always start at the same position. There are also the option of advancing or retarding the pulse in a slice.

When working with synchronized PWM signals, we may want the pulses to have not the same start, but have the same center, so they remain in phase when the duty cycle is changed. For this there is a *phase-correct* mode. In this mode the counter starts at zero, counts up until the wrap value, counts down to zero, and repeats. The output will change to low when the compare level is reached in the up count and change to high when the compare level is reached in the down count:



Phase Correct PWM waveforms



Phase Correct counter operation

Note that the frequency for phase-correct will be half of the frequency for not phase-correct (for the same divisor and wrap value).

Measuring Frequency and Duty Cycle

For measurement we will choose another counter option instead of the free-running. When this is done, the B pin changes to input and the corresponding compare level is ignored.

Let's start by looking at duty cycle measurement. The idea here is that we will sample the input signal at the frequency of f_{sys} and generate a clock pulse only if the signal is high. This clock pulses will be divided by the clock divisor and increment the counter. The steps to do the measurement are:

- We select the pin we will use and look up to what slice and PWM pin it can be connected, making sure it is a B pin.
- We configure the clock mode to level sensitivity
- We configure the divisor
- We set the function of the pin to PWM
- We let the counter run for a known time
- We read the counter value and calculate the duty cycle

For example, let's say we are using the default f_{sys} of 125 MHz, configure the divisor for 200 and get a count of 625:

- In 10 ms we have $125,000,000 * 0.01 = 1250,000$ f_{sys} pulses
- The maximum possible count (duty cycle 100%) is $1250,000 / 200 = 6250$
- 625 out of 6250 corresponds to 10%

So the duty cycle for count c after running for time t with a divisor d is $100 * c * d / (t * f_{sys})$.

When choosing the divisor, the ideal is to get the maximum possible count (for precision) that will not overflow the 16 bits of the counter. In my example a divisor of 20 would give a better resolution, but 6250 values for duty cycle is more than enough.

Frequency measurement is done by counting the rising or falling edges (changes in the level) in the input signal. Each edge is a clock pulse that goes through the clock divider and increment the counter. The low and high times of the signal must be greater than the f_{sys} period for edge detection to work. The steps required are:

- We select the pin we will use and look up to what slice and PWM pin it can be connected, making sure it is a B pin.
- We configure the clock mode to edge sensitivity
- We configure the divisor
- We set the function of the pin to PWM
- We let the counter run for a known time
- We read the counter value and calculate the frequency

For example if the divisor is 10 and we get a count of 3000 in 0.1 second:

- The count of 3000 indicates we had 30,000 edges in 0.1 second
- That corresponds to 300,000 cycles per second (300 kHz)

So the frequency for count c after running for time t with a divisor d is $c * d / t$. For better precision we should use lower d and/or greater t (taking care to not overflow the counter).

Selected SDK Functions

These functions are in the library `hardware_pwm`. The slices are numbered from 0 to 7.

Pin Association

```
static uint pwm_gpio_to_slice_num (uint gpio)
```

Returns the number of the PWM slice connected to a gpio pin.

```
static uint pwm_gpio_to_channel (uint gpio)
```

Returns the channel of the PWM slice connected to a gpio pin:

- PWM_CHAN_A (0)
- PWM_CHAN_B (1)

Slice Configuration - Set 1

This first set of configuration routines change the slice directly.

```
static void pwm_set_clkdiv (uint slice_num, float divider)
```

Changes the clock divisor in the slice to the binary equivalent of div.

```
static void pwm_set_output_polarity (uint slice_num, bool a, bool b)
```

Changes the output polarity of both channels of a slice:

- a true inverts output A
- b true inverts output B

```
static void pwm_set_clkdiv_mode (uint slice_num, enum pwm_clkdiv_mode mode)
```

Changes the counter mode of a slice. Options for mode are:

- PWM_DIV_FREE_RUNNING selects free-running mode, channels A and B are outputs.
- PWM_DIV_B_RISING selects rising edge sensitivity, channel B is input.
- PWM_DIV_B_FALLING selects falling edge sensitivity, channel B is input.
- PWM_DIV_B_HIGH selects high level sensitivity, channel B is input.

```
static void pwm_set_phase_correct (uint slice_num, bool phase_correct)
```

Changes the phase correct option in a slice. `phase_correct` true enables phase correct, false disables.

```
static void pwm_set_wrap (uint slice_num, uint16_t wrap)
```

Sets the wrap value in a slice.

```
static void pwm_set_chan_level (uint slice_num, uint chan, uint16_t level)
```

Sets the level of a channel in a slice. Options for channel are PWM_CHAN_A and PWM_CHAN_B.

```
static void pwm_set_both_levels (uint slice_num, uint16_t level_a, uint16_t level_b)
```

Sets the level of both channels in a slice.

```
static void pwm_set_gpio_level (uint gpio, uint16_t level)
```

Looks up the slice and channel connected to a gpio pin and set its level.

```
static void pwm_set_enabled (uint slice_num, bool enabled)
```

Enables or disables a slice.

```
static void pwm_set_mask_enabled (uint32_t mask)
```

Enable/Disable multiple PWM slices simultaneously. Bits 0 to 7 of mask corresponds to slices 0 to 7. A value 0 in a bit disables the slice, a value 1 enables it.

Slice Configuration - Set 2

In this second set of configuration routines an struct (pwm_config) is used to set up the configuration. Use `pwm_config pwm_get_default_config()` to get an initialized struct, change it with the `pwm_config_set_xxx` functions and then apply it to a slice using `pwm_init()`.

Note: The levels are not part of the configuration structure.

```
static pwm_config pwm_get_default_config (void)
```

Returns an initialized configuration structure.

```
static void pwm_config_set_phase_correct (pwm_config *c, bool phase_correct)
```

Changes the phase correct option in the configuration (phase_correct true enables phase correct, false disables).

```
static void pwm_config_set_clkdiv (pwm_config *c, float div)
```

Changes the clock divisor in the configuration to the binary equivalent of div.

```
static void pwm_set_clkdiv_int_frac (uint slice_num, uint8_t integer, uint8_t fract)
```

Changes the clock divisor in the configuration.

```
static void pwm_config_set_clkdiv_int (pwm_config *c, uint div)
```

Changes the clock divisor in the configuration to div, with zero in the fractional part.

```
static void pwm_config_set_clkdiv_mode (pwm_config *c, enum pwm_clkdiv_mode mode)
```

Changes the counter mode in the configuration. Options for mode are:

- PWM_DIV_FREE_RUNNING selects free-running mode, channels A and B are outputs.
- PWM_DIV_B_RISING selects rising edge sensitivity, channel B is input.
- PWM_DIV_B_FALLING selects falling edge sensitivity, channel B is input.

- PWM_DIV_B_HIGH selects high level sensitivity, channel B is input.

```
static void pwm_config_set_output_polarity (pwm_config *c, bool a, bool b)
```

Changes the output polarity of both channels in the configuration:

- a true inverts output A
- b true inverts output B

```
static void pwm_config_set_wrap (pwm_config *c, uint16_t wrap)
```

Sets the wrap value in the configuration.

```
static void pwm_init (uint slice_num, pwm_config *c, bool start)
```

Initializes a slice as specified by the configuration. If start is true, the slice is enabled, otherwise you will have to enable it with `pwm_set_enabled()` or `pwm_set_mask_enabled()`.

Counter Manipulation

```
static uint16_t pwm_get_counter (uint slice_num)
```

Returns the current counter value for a slice.

```
static void pwm_set_counter (uint slice_num, uint16_t c)
```

Sets the counter for a slice.

```
static void pwm_advance_count (uint slice_num)
```

Advances the counter of a running slice by inserting a clock pulse after the current one. This requires a divisor greater than one. This function blocks until the extra clock pulse is started.

```
static void pwm_retard_count (uint slice_num)
```

Retards the counter of a running slice, by canceling the next clock pulse. This function blocks until the canceled clock pulse starts.

Interrupt and DMA

```
static void pwm_set_irq_enabled (uint slice_num, bool enabled)
```

Enables or disables interrupt request by a slice.

```
static void pwm_set_irq_mask_enabled (uint32_t slice_mask, bool enabled)
```

Enables or disables interrupt requests in the slice selected by mask (bit 0 to 7 corresponds to slice 0 to 7, values '1' mark the slices affected).

```
static void pwm_clear_irq (uint slice_num)
```

Clears interrupt request of a slice.

```
static void pwm_force_irq (uint slice_num)
```

Forces an interrupt request by a slice.

```
static uint32_t pwm_get_irq_status_mask (void)
```

Bits 0 to 7 indicate if interrupt is enabled in slices 0 to 7. A value of '1' indicates the interrupt is enabled, '0' indicates disabled.

```
static uint pwm_get_dreq (uint slice_num)
```

Returns the DMA request number for a slice.

Examples

PWM Generation

This example was used to generate the waveforms presented earlier. Frequency is 500Hz, duty cycle is 50% for pin A and 25% for pin B. To use the examples as coded you need to connect the RP2040 board to a PC via USB and connect to it using a serial communication program (like puTTY or the Arduino IDE monitor).

PWM generation will not start until there is a connection. Sending any character to the board will change the waveforms between phase correct and normal PWM.

PWM Generation Example

```

1  /**
2   * @file pwmdemo.c
3   * @author Daniel Quadros
4   * @brief Example of using the PWM in the RP2040
5   *       This example was used to generate the figures in the boot
6   * @version 0.1
7   * @date 2022-07-09
8   *
9   * @copyright Copyright (c) 2022, Daniel Quadros
10  *
11  */
12
13  #include <stdio.h>
14  #include <string.h>
15  #include <stdlib.h>
16
17  #include "pico/stdlib.h"
18  #include "hardware/clocks.h"
19  #include "hardware/gpio.h"
20  #include "hardware/pwm.h"
21
22  // PWM pins
```

```
23 #define PIN_A 0
24 #define PIN_B 1
25
26 // WRAP value
27 #define WRAP 1000
28
29 // Frequency
30 #define FREQ 500.0f
31
32 // Main Program
33 int main() {
34     // Init stdio
35     stdio_init_all();
36     while (!stdio_usb_connected()) {
37         sleep_ms(100);
38     }
39     printf("\nPWM Example\n");
40
41     // Find out which PWM slice is connected to the pins
42     uint slice_num = pwm_gpio_to_slice_num(PIN_A);
43     if (slice_num != pwm_gpio_to_slice_num(PIN_B)) {
44         printf("Pins are not in the same slice!\n");
45         printf("Aborting...\n");
46         while (true) {
47             sleep_ms(100);
48         }
49     }
50
51     // Configure the slice
52     // f = fsys / (clock divisor * (wrap value+1))
53     // clock divisor = fsys / (f * (wrap value+1))
54     float fsys = frequency_count_khz(CLOCKS_FC0_SRC_VALUE_CLK_SYS)*1000.0f;
55     float div = fsys/(FREQ * (WRAP+1));
56     printf("fsys= %.2f div=%.2f\n", fsys, div);
57     pwm_config config = pwm_get_default_config ();
58     pwm_config_set_wrap(&config, WRAP);
59     pwm_config_set_clkdiv(&config, div);
60     pwm_config_set_phase_correct(&config, false);
61     pwm_config_set_clkdiv_mode(&config, PWM_DIV_FREE_RUNNING);
62     pwm_init(slice_num, &config, false);
63     pwm_set_both_levels(slice_num, WRAP/2, WRAP/4);
64     pwm_set_enabled(slice_num, true);
65 }
```

```

66     // Connect PINs to the PWM
67     gpio_set_function(PIN_A, GPIO_FUNC_PWM);
68     gpio_set_function(PIN_B, GPIO_FUNC_PWM);
69
70     // Main loop
71     bool phase_correct = false;
72     while (true) {
73         if (getchar_timeout_us(0) != PICO_ERROR_TIMEOUT) {
74             // Change phase correct if anything received from stdio
75             // Stop PWM while changing configuration
76             // If phase correct, PWM will count twice,
77             // so we double the clock frequency
78             phase_correct = !phase_correct;
79             pwm_set_enabled(slice_num, false);
80             pwm_set_clkdiv(slice_num, phase_correct? div/2.0f : div);
81             pwm_set_phase_correct(slice_num, phase_correct);
82             pwm_set_counter(slice_num, 0);
83             pwm_set_enabled(slice_num, true);
84             printf ("Phase correct: %s\n", phase_correct? "ON" : "OFF");
85         }
86     }
87 }

```

Frequency and Duty-cycle Measurement

For this example you need to connect GP1 to GP3. The software will generate various PWM signals and try to measure them.

PWM Measurement Example

```

1  /**
2   * @file pwmmesurement.c
3   * @author Daniel Quadros
4   * @brief Example of using the PWM peripheral in the RP2040 for
5   *        measuring frequency and duty cycle
6   *        This is an expansion of the measure_duty_cycle SDK example
7   * @version 0.1
8   * @date 2022-07-11
9   *
10  * @copyright Copyright (c) 2022, Daniel Quadros
11  *
12  */
13
14  #include <stdio.h>

```

```
15 #include <string.h>
16 #include <stdlib.h>
17
18 #include "pico/stdlib.h"
19 #include "hardware/clocks.h"
20 #include "hardware/gpio.h"
21 #include "hardware/pwm.h"
22
23 // Pins - this pins should be connected together
24 const uint OUTPUT_PIN = 1;
25 const uint MEASURE_PIN = 3;    // this must be an PWM "B" pin
26
27 // WRAP value for PWM generation
28 #define WRAP 1000
29
30 // Values for measurement
31 #define MEASURE_C_DIV    20
32 #define MEASURE_C_TIME   10    // ms
33 #define MEASURE_F_DIV    1
34 #define MEASURE_F_TIME   100   // ms
35
36 // Test values
37 struct {
38     float freq;
39     float duty;
40 } test[] =
41 {
42     { 500.0f, 0.0f },
43     { 500.0f, 1.0f },
44     { 500.0f, 0.25f },
45     { 500.0f, 0.5f },
46     { 500.0f, 0.75f },
47     { 492.0f, 0.60f },
48     { 947.0f, 0.60f },
49     { 1000.0f, 0.25f },
50     { 0.0, 0.0 }
51 };
52
53 // Generate PWM
54 void generate_pwm(int slice, float freq, float duty) {
55     float fsys = clock_get_hz(clk_sys);
56     float div = fsys/(freq * (WRAP+1));
57     pwm_config config = pwm_get_default_config ();
```


```

58     pwm_config_set_wrap(&config, WRAP);
59     pwm_config_set_clkdiv(&config, div);
60     pwm_config_set_phase_correct(&config, false);
61     pwm_config_set_clkdiv_mode(&config, PWM_DIV_FREE_RUNNING);
62     pwm_init(slice, &config, false);
63     pwm_set_chan_level(slice, pwm_gpio_to_channel(OUTPUT_PIN), (uint16_t) (duty*(WRA\
64 P+1)));
65     pwm_set_enabled(slice, true);
66 }
67
68 // Measure frequency
69 float measure_frequency(uint slice) {
70
71     // Count once for every MEASURE_DIV cycles the PWM B input is high
72     pwm_config cfg = pwm_get_default_config();
73     pwm_config_set_clkdiv_mode(&cfg, PWM_DIV_B_RISING);
74     pwm_config_set_clkdiv(&cfg, MEASURE_F_DIV);
75     pwm_init(slice, &cfg, false);
76     gpio_set_function(MEASURE_PIN, GPIO_FUNC_PWM);
77
78     // This is where the actual count is done
79     pwm_set_enabled(slice, true);
80     sleep_ms(MEASURE_F_TIME);
81     pwm_set_enabled(slice, false);
82
83     // Calculate frequency
84     return (pwm_get_counter(slice) * MEASURE_F_DIV * 1000.0f) / MEASURE_F_TIME;
85 }
86
87 // Measure duty cycle
88 float measure_duty_cycle(uint slice) {
89
90     // Count once for every MEASURE_DIV cycles the PWM B input is high
91     pwm_config cfg = pwm_get_default_config();
92     pwm_config_set_clkdiv_mode(&cfg, PWM_DIV_B_HIGH);
93     pwm_config_set_clkdiv(&cfg, MEASURE_C_DIV);
94     pwm_init(slice, &cfg, false);
95     gpio_set_function(MEASURE_PIN, GPIO_FUNC_PWM);
96
97     // This is where the actual count is done
98     pwm_set_enabled(slice, true);
99     sleep_ms(MEASURE_C_TIME);
100    pwm_set_enabled(slice, false);

```

```
101
102     // Calculate duty cycle
103     float counting_rate = clock_get_hz(clk_sys) * ((float) MEASURE_C_TIME / 1000.0f);
104     float max_possible_count = counting_rate / MEASURE_C_DIV;
105     return pwm_get_counter(slice) / max_possible_count;
106 }
107
108 // Main Program
109 int main() {
110     // Init stdio
111     stdio_init_all();
112     while (!stdio_usb_connected()) {
113         sleep_ms(100);
114     }
115     printf("\nPWM Measurement Example\n");
116
117     // Find out which PWM slice is connected to the pins
118     uint slice_out = pwm_gpio_to_slice_num(OUTPUT_PIN);
119     uint slice_mea = pwm_gpio_to_slice_num(MEASURE_PIN);
120
121     assert(pwm_gpio_to_channel(MEASURE_PIN) == PWM_CHAN_B);
122
123     // Connect PINS to the PWM
124     gpio_set_function(OUTPUT_PIN, GPIO_FUNC_PWM);
125
126     // Main loop
127     while (true) {
128         for (int i = 0; test[i].freq != 0.0; i++) {
129             generate_pwm(slice_out, test[i].freq, test[i].duty);
130             float freq = measure_frequency(slice_mea);
131             float duty = measure_duty_cycle(slice_mea);
132             pwm_set_enabled(slice_out, false);
133             printf ("Freq %.2f x %.2f   Duty %.2f x %.2f\n",
134                 test[i].freq, freq, test[i].duty, duty);
135         }
136         while (getchar_timeout_us(0) == PICO_ERROR_TIMEOUT) {
137             sleep_ms(100);
138         }
139         printf("\n");
140     }
141 }
```

Take a look at the results:

 COM24 - PuTTY

```
PWM Measurement Example
Freq  500.00 x 0.00    Duty 0.00 x 0.00
Freq  500.00 x 0.00    Duty 1.00 x 1.00
Freq  500.00 x 500.00   Duty 0.25 x 0.25
Freq  500.00 x 500.00   Duty 0.50 x 0.50
Freq  500.00 x 500.00   Duty 0.75 x 0.75
Freq  492.00 x 490.00   Duty 0.60 x 0.59
Freq  947.00 x 940.00   Duty 0.60 x 0.59
Freq  1000.00 x 1000.00  Duty 0.25 x 0.25
```

PWM Measurement Output

Notice that for 0 and 100% duty-cycles the frequency is zero - the signal does not change! We can also see that our precision in measurements is not perfect. For example, a frequency of 947Hz is measure as 940Hz. This happens because we are looking at the signal for just 0.1 second so we will see only 94 rising edges. To get a better precision of a frequency in this range we would need to increase the counting time.

The Programmable I/O (PIO)

The PIO is a kind of peripheral that you will not find in most microcontrollers. It is a solution for interfacing challenges normally solved by manual control of I/O and precise timings that do not integrate nicely with other software requirements.

You will find out that a lot of space in the official RP2040 documentation is dedicated to the PIO, but it may still sound mysterious and complicated. I will do my best to guide you through an easy path to understanding this important part of the RP2040.

The PIO State Machines

The basic unit of the PIO are the **state machines**. They are small processors that execute small programs concurrently with the execution of the two ARM cores. This means that they can keep doing their jobs without interfering with the execution of the main firmware, and vice-versa.

The state machines communicate with the ARM core using queues (**FIFOs** - First In First Out) and interrupts. The FIFOs can also work with the DMA controller, with data moving directly between the FIFOs and memory.

The State Machines can interact with GPIOs, not only doing digital input and output but also controlling the direction of the pins (changing them between input and output during execution).

The RP2040 has two PIOs, each with four State Machines. Each PIO has also a 32 position **instruction memory** that is shared by the four State Machines. The instruction memory store the programs that the state machines will execute.

The FIFOs

Each State Machine has two FIFOs, each with four 32 bit positions. Normally one FIFO is for receiving (send data from the PIO to the processor) and the other for transmitting (sending data from the processor to the PIO).

If the PIO will only receive or transmit, the FIFOs can be configured as a single eight position queue.

The function of the FIFOs is to allow the PIOs to communicate asynchronously with the main CPUs instead of requiring their intervention as each word is received or sent.

Programmer's Model

When writing a program for a State Machine, the Programmer will use five 32 bit registers:

- **Out Shift Register (OSR):** this register will get data from the TX FIFO and shift it to the right or left, inserting zeros on the other side. The shifted out bits can be outputted to GPIOs.
- **In Shift Register (ISR):** this register will shift bits to the right or left and put the result to the RX FIFO. The shifted in bits can be inputted from GPIOs.
- **Scratch Registers (X and Y):** this registers can be used as source or destination for some instructions.
- **Program Counter (PC):** this register points to the current executing instruction and can be used as destination in a few instructions.

PIO Configuration

Part of the complexity in understanding the PIOs is that the behavior of the state machines depends not only on the programs they are executing but on the way they are configured. Let's see what can be configured.

GPIO Pins Mapping

In most applications we will want the PIO to do input and output through GPIO pins.

The RP2040 controls the GPIOs through 32 bit registers, where each bit corresponds to a GPIO - actually the RP2040 has only 30 GPIOs (GP00 to GP29), so two bits are unused. The state machines do not use this numbers. The GPIOs must be mapped into the numbers used in the PIO programming.

The idea behind the mappings is to align the pins to the bits in the state machine registers. This is done by defining a “base pin” that will correspond to bit 0 (lowest bit) in the registers. In this mapping, GP00 is considered next to GP32, so the mapping can “wrap around”. For example, if you configure a base pin to 9, bit 0 will be associated with GP09, bit 1 to GP10 and so on.

The configuration allows us to specify five groups of pins, each group used in a different situation:

- **Input:** we configure the “base pin” that will be input pin 0 for the state machine. This mapping is used by the WAIT, IN and MOV (for the source) instructions. Input pins are normally used to receive data that will be put in the RX FIFO or to wait a change on a pin.
- **Output:** we configure the “base pin” and the number of pins that will be used for output. This mapping is used by the OUT and MOV (for the destination) instructions. Output pins are normally used to send data from the TX FIFO.
- **Set:** we configure the “base pin” and the number of pins that will be used for the SET instruction. Set pins are used when the change of a pin does not come directly from the data in the TX FIFO.

- **Side-Set:** we configure the “base pin” and number of side-set pins (up to five). We can also configure a side-set enable bit, that will control if side-set is used on each instruction. Side-set pins can be changed by any instruction but, as described in the next section, using side-pins will limit the delay that can be used in the instructions. Side-set is used when a pin must change at the same time a instruction is executed.
- **Jump-Pin:** one pin can be configured to be tested by the JMP instruction. The jump-pin is used when you need to change the flow of the program based on a pin.

Of course the uses mentioned above not a fixed rule, you can use your imagination and find other uses for the GPIO mapping, as long as you respect the groups used by each instruction.

When an output or side set pin is used, we can change the state of the pin (low or high) or its direction (input or output).

Clock

Each state machine can have a different clock, created by dividing the `clk_sys` (125MHz, if you haven't changed it) by a *fractional divider*. This divider has 24 bits, 16 for the integer part and 8 for the fractional part (in units of 1/256).

For example, if we want to use a clock of 50MHz, we need to divide `clk_sys` by 2.5 (that is an integer part of 2 and a fractional part of 128).

This clock will define the **cycle time**. Every PIO instruction can be executed in one cycle, but you can add an extra delay.

The maximum delay available will depend on the configuration for the side-set pins, as both features share the same 5 bit field in the instruction code. If you are not using side-set, you can specify up to 31 delay cycles. If you have 1 side-set pin, the maximum goes down to 15, and so on.

If you need to generate precision timing you will have to choose carefully the divider and take into account the instructions execution time (including the delay) in all execution paths. We will see how it is done in the examples.

FIFOs Configuration

There are a few configurations that affect the FIFOs, some of them will affect how the program will operate.

The first configuration allows to join the two FIFOs into a single RX or TX FIFO. This is useful when you are only receiving or transmitting.

The next configurations are the auto-pull and auto-push. Remember that the output shift register (OSR) is fed by the TX FIFO and that the RX FIFO is fed by the input shift register (ISR)? There are two ways to do the moving between the shift registers and the FIFOs:

- You can do that explicitly using the PUSH and PULL instructions.
- It can be done automatically by activating auto-pull or auto-push. In both cases you also set how many bits need to be shift to do the moving.

When doing serial communications the auto options will simplify your code, as you will not need to count how many bits were shifted.

Program Wrapping

Later we talk more about the control flow in PIO programs, but there is a special case that's implemented as a configuration.

Most PIO programs will loop forever: when they reach the end of the program they need to jump back to a previous instruction. This can be done with a JMP instruction, but it will cost one instruction and one cycle.

Each state machine has two configuration registers (EXECCTRL_WRAP_TOP and EXECCTRL_WRAP_BOTTOM). After executing the instruction at EXECCTRL_WRAP_TOP (if its not a JMP that is taken) execution will proceed at EXECCTRL_WRAP_BOTTOM instead of the next instruction (with no time penalty).

This configuration can be done by placing two special directives in the source program (.wrap and .wrap_target).

Interrupt (IRQ) Flags

There are eight IRQ Flags available to all state machines, numbered 0 to 7. The lower four (0 to 3) can be associated to one of the two PIO's interrupt request lines.

One use for the IRQ Flags is to generate an interrupt to notify one of the ARM cores. The PIO program can not only set a IRQ flag but also wait for the interrupt to be acknowledge by one of the cores.

Another use is to synchronize two state machines, as the flags are shared by all.

The Instructions

The table bellow shows the complete PIO instruction set.

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JMP	0	0	0	Delay/side-set					Condition			Address				
WAIT	0	0	1	Delay/side-set					Pol	Source		Index				
IN	0	1	0	Delay/side-set					Source			Bit count				
OUT	0	1	1	Delay/side-set					Destination			Bit count				
PUSH	1	0	0	Delay/side-set					0	IfF	Blk	0	0	0	0	0
PULL	1	0	0	Delay/side-set					1	IfE	Blk	0	0	0	0	0
MOV	1	0	1	Delay/side-set					Destination			Op		Source		
IRQ	1	1	0	Delay/side-set					0	Clr	Wait	Index				
SET	1	1	1	Delay/side-set					Destination			Data				

PIO Instructions

Each instruction execute in one clock cycle and uses 16 bits in the program memory:

- The first three bits determine the instruction.
- The next five bits are divided between delay and side-set. As we saw, you can configure from 0 to 5 side-set pins (optionally including the side-set enable bit). The delay encoded in the remaining bits is the number of clock cycles waited after the instruction executes, before executing the next instruction.
- The remaining eight bits encodes the operands of the instruction.

In the following assembly codings, (x) means that x is optional and {x} represents an expression that will result in x. Side set and delay values can be added to any of the instructions:

```
{instruction} (side {side_set_value}) ([{delay_value}])
```

JMP

The format of the JMP opcode is

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JMP	0	0	0	Delay/side-set					Condition			Address				

PIO JMP Instruction

The JMP instruction set the program counter to Address if Condition is satisfied. The delay takes effect after the instruction executes, regardless of the jumping taking place.

The condition available are:

- 000 / (no condition): Always
- 001 / !X: register X is zero
- 010 / X-: register X is not zero, after the test X is always decremented
- 011 / !Y: register Y is zero
- 100 / Y-: register Y is not zero, after the test Y is always decremented
- 101 / X!=Y: register X not equal to Y
- 110 / PIN: true if input pin is 1
- 111 / !OSRE: output shift register not empty (at least SHIFTCTRL_PULL_THRESH bits were shifted into the OSRE since last PUSH or auto-push)

The coding of the JMP instruction in assembly is as follows:

```
jmp (cod) {target}
```

cod is the optional condition (!X, X-, !Y, Y-, X!=Y, PIN, !OSRE)

target is the a program label or address. While the encoding uses an absolute address, in assembly we use a value relative to the start of the program (the assembler takes care of adding the start address).

WAIT

The format of the WAIT opcode is:

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
WAIT	0	0	1	Delay/side-set					Pol	Source		Index				

PIO WAIT Instruction

This instruction **stalls** (stops) execution until a condition is met. The side-set (if used) is done when the instruction starts execution, the delay starts after the condition is met.

Source specify what we are waiting for:

- 00 (GPIO): GPIO selected by index (this does not go through input pins mapping)
- 01 (PIN): Input pin selected by index (according o the input pins mapping)
- 10 (IRQ): PIO IRQ selected by index: if the most significant bit (MSB) is 0, the lower 2 bits of index select the IRQ flag; if MSB is 1, the state machine number (0 to 3) is add to index and the lower 2 bits of the result select the IRQ flag.
- 11 (RESERVED): not used

For GPIO and PIN, Pol (polarity) determines what value we are waiting for (0 or 1). For IRQ, pol = 1 means clear IRQ after condition is met.

The WAIT instruction is coded in assembly as follows:

```
wait {pol} gpio {gpio_num}
wait {pol} pin {pin_num}
wait {pol} irq {irq_num} (rel)
```

The `rel` in the third option sets the MSB in the index, making `irq_num` “relative” to the state machine.

IN

The format format of the IN opcode is:

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IN	0	1	0	Delay/side-set				Source			Bit count					

PIO IN Instruction

The IN instruction will shift `bit count` bits from the `source` into the ISR:

- The least significant `bit count` bits of the source will be used as input (regardless of the configured shift direction).
- The ISR is shifted `bit count` bits in the configured direction and the source bits are put in the “opened” positions.
- The input shift count is increase by `bit count` (stopping at 32).
- If auto push is enabled and the configured threshold is reached, the ISR is pushed into the Rx FIFO and cleared to zeros (the input shift count is also zeroed). The state machine stalls if there is no space in the FIFO (execution resumes when the push can be done).

The available sources are:

- 000 (PINS): `bit count` pins (using the configured input pin mapping)
- 001 (X): X register
- 010 (Y): Y register
- 011 (NULL): zeros
- 100: reserved
- 101: reserved
- 110 (ISR)
- 111 (OSR)

A `bit count` of zero is treated as 32.

The IN instruction is coded in assembly as follows:

```
in {source},{bit_count}
```

OUT

The format of the OUT opcode is:

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OUT	0	1	1	Delay/side-set					Destination			Bit count				

PIO OUT Instruction

The OUT instruction will shift `bit_count` bits from the OSR into the destination:

- A zero filled 32 bit value will be written to `destination`, with the least significant bits coming from:
 - if the shift direction is to the right, the least significant `bit_count` in the OSR
 - if the shift direction is to the left, the most significant `bit_count` in the OSR
- The output shift count is increased by `bit_count` (stopping at 32)
- If auto pull is enabled and the configured threshold is reached, the OSR is filled from the Tx FIFO and the output shift count is zeroed. The state machine stalls if there is no data in the FIFO (execution resumes when the pull can be done).

The available sources are:

- 000 (PINS): `bit_count` pins (using the configured output pin mapping)
- 001 (X): X register
- 010 (Y): Y register
- 011 (NULL): zeros
- 100 (PINDIR): set direction of `bit_count` pins (using the configured output pin mapping)
- 101 (PC): causes a jump
- 110 (ISR): also sets input shift register counter to `bit_count`
- 111 (EXEC): execute data as instruction

A `bit_count` of zero is treated as 32.

The OUT instruction is coded in assembly as follows:

```
out {destination},{bit_count}
```


PUSH

The format of the PUSH opcode is:

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PUSH	1	0	0	Delay/side-set					0	IfF	Blk	0	0	0	0	0

PIO PUSH Instruction

Operation of the PUSH instruction depends on two flags in the opcode:

- If *IfFull* is 1, the instruction does nothing if the input shift counter has not reached its threshold. If it is 0 or the threshold was reached, continues as follows
- If *Block* is 0 and the Rx FIFO is full, ISR (and its counter) is cleared to zero and execution proceeds with the next instruction. If *Block* is 1 and the Rx FIFO is full, state machine stalls until there is space in the FIFO, then continues. If the Rx FIFO is not full, continues
- Put the content of ISR in the Rx FIFO and clear ISR (and its counter) to zero

The PUSH instruction is coded in assembly as follows:

```
push (iffull)
push (iffull) block
push (iffull) noblock
```

As the defaults are *IfFull* = 0 and *Block* = 1, you will normally just use `push`. *IfFull* should be used only if you cannot use automatic push (because it could block some instruction). *Block* = 0 should be used in a context where blocking is worse than losing data if the FIFO becomes full.

PULL

The format of the PULL opcode is:

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PULL	1	0	0	Delay/side-set					1	IfE	Blk	0	0	0	0	0

PIO PULL Instruction

Operation of the PULL instruction depends on two flags in the opcode:

- If *IfEmpty* is 1, the instruction does nothing if the output shift counter has not reached its threshold. If it is 0 or the threshold was reached, continues as follows
- If *Block* is 0 and the Tx FIFO is empty, register X is copied into OSR, its counter is cleared and execution proceeds with the next instruction. If *Block* is 1 and the Tx FIFO is empty, state machine stalls until there is data in the FIFO, then continues. If the Tx FIFO is not empty, continues

- Pull the top of Tx FIFO into OSR and set its counter to zero

The PULL instruction is coded in assembly as follows:

```
push (ifempty)
push (ifempty) block
push (ifempty) noblock
```

As the defaults are IfEmpty = 0 and Block = 1, you will normally just use `pull`. IfEmpty should be used only if you cannot use automatic pull (because it could block some instruction). Block = 0 should be used in a context where a default value should be used if there is no data available in the Tx FIFO.

MOV

The format of the MOV opcode is:

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOV	1	0	1	Delay/side-set				Destination				Op		Source		

PIO MOV Instruction

The MOV instruction copies data from a source to a destination, applying operation.

Source can be:

- 000 (PINS): uses input pin mapping
- 001 (X)
- 010 (Y)
- 011 (NULL): zeros
- 100 Reserved
- 101 (STATUS): all zeros or all ones, depending on state machine status configured by `EXECCTRL_STATUS_SEL`
- 110 (ISR)
- 111 (OSR)

Destination can be:

- 000 (PINS): uses output pin mapping
- 001 (X)
- 010 (Y)
- 011 Reserved
- 100 (EXEC): execute as instruction (ignores delay in original MOV instruction, uses delay in EXEC'd instruction)

- 101 (PC): causes a jump
- 110 (ISR): ISR counter set to zero
- 111 (OSR): OSR counter set to zero

The operations available are:

- 00: None, copy data unchanged
- 01: Invert bits (0->1 and 1->0)
- 10: Reverse the data (bit n <-> bit 31-n)
- 11: Reserved

The MOV instruction is coded in assembly as follows:

```
mov {destination}, (op), {source}
```

Omit op for “None”, use ! or ~ for “Invert” and use :: for “Reverse”

IRQ

The format of the IRQ opcode is:

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IRQ	1	1	0	Delay/side-set					0	Clr	Wait	Index				

PIO IRQ Instruction

The IRQ instruction is used to manipulate the IRQ Flags. The operation depends on the Clear and Wait bits in the opcode:

Clear	Wait	Operation
0	0	Set an IRQ Flag
0	1	Set an IRQ Flag and wait until it is cleared
1	x	Clear an IRQ Flag (Wait is ignored)

The IRQ Flag is selected by the index field:

- If the MSB of the index is zero, the lower three bits select the flag.
- If the MSB of the index is one, the lower three bits are added to the state machine index and the lower three bits of the result select the flag. This is useful if the same code will be run by more than one state machine and different flags should be used.

The IRQ instruction is coded in assembly as follows:

```

irq {irq_num} (_rel)
irq set {irq_num} (_rel)
irq nowait {irq_num} (_rel)
set without waiting

irq wait {irq_num} (_rel)
set and wait

irq clear {irq_num} (_rel)
clear

```

SET

The format of the SET opcode is:

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SET	1	1	1	Delay/side-set					Destination			Data				

PIO SET Instruction

The SET instruction writes an immediate value (0 to 31) into a destination.

Destination can be:

- 000 (PINS): uses set pin mapping
- 001 (X)
- 010 (Y)
- 011 Reserved
- 100 (PINDIRS): uses set pin mapping
- 101 Reserved
- 110 Reserved
- 111 Reserved

The SET instruction is coded in assembly as follows:

```
set {destination},{value}
```

Flow Control

Typically instructions for a PIO State Machine comes from its program memory.

The State Machine Program Counter (PC) points to the executing instruction. After the instruction completes, it is updated to the next instruction. This is normally in the next address in the program

memory, but JMP, IN and MOV instructions can change the PC. There is also the program wrap that will make the program jump when it reach an address.

An instruction takes at least one cycle to execute. WAIT, IN, OUT, PUSH, PULL and IRQ instructions can **stall** - stop execution until a condition is met. After the instruction executes we can have an additional delay specified in the delay/side-set field.

There are three ways to execute PIO instructions from outside the program memory:

- Write the instruction in the special configuration register SMx_INSTR
- Use MOV EXEC to execute an instruction that is in a register
- Use OUT EXEC to execute an instruction in the output shifter

The OUT EXEC option can be used to mix instructions and data in the values written to the FIFO.

Coding, Compiling and Running PIO Programs

The preferred way to code a PIO program for use with the C/C++ SDK is to create a .pio file. This file will contain:

- The PIO program, written in the PIO assembly language
- A C routine to initialize/configure a state machine for execution of the PIO program.

The PIO assembler will translate the .pio file into a C header file (.h) for inclusion in the C source where the program will be used. Assuming you have included `pico_sdk_import.cmake` and called `pico_sdk_init()` in your `CMakeLists.txt`, you will call the assembler in the `CMakeLists.txt` by adding the line

```
pico_generate_pio_header(project ${CMAKE_CURRENT_LIST_DIR}/xxx.pio)
```

where `project` is the name of your project and `xxx.pio` your source pio file.

The PIO assembler can also target other languages like MicroPython. We will not cover that in this book.

PIO Assembly Language

In the previous section we have seen the assembly coding for the instructions. Let's take now a closer look at the full assembly language.

Directives

Directives are statements that control the assembly and execution of the PIO program but are not translated to PIO instructions. All instructions start with a dot.

`.program {name}`

Start a program. PIO instructions can only be used inside a program.

`.define (PUBLIC) {symbol} {value}`

Associates value to symbol. If the define is before the first program in file, it can be used in all programs in the file. Otherwise it can be used in the program it occurs.

If PUBLIC is used, a definition will be included in the C header file as `#define <program_name>_-<symbol> value`

`.origin {offset}`

Defines the position in the instruction memory where the following instructions will run. Must be used inside a program.

`.side_set {count} (opt) (pindirs)`

This directive configures the use of side set and must be used inside a program before any instruction. `count` is the number of bits to be reserved for side set. If `opt` is used, the side set is optional (an additional bit will be used). `pindirs` indicate that the side set will affect PINDIRS instead of PINS.

`.wrap_target`

This marks the place where execution will go when it wraps. Must be inside a program and only one can be used for each program. If not used, the default is the beginning of the program.

`.wrap`

This marks where the program will wrap (jump to the wrap target). Must be inside a program and only one can be used for each program. If not used, the default is the end of the program.

`.lang_opt {lang} {name} {option}`

This sets an option for a specific language translator.

`.word {value}`

Insert a 16 bit value as an instruction. Must be used inside a program.

Values and Expressions

A value can be one of the following:

- an integer number, like -1 and 31
- an hexadecimal number, prefixed by 0x, like 0x42

- a binary number, prefixed by 0b, like 0b1010101
- a symbol, define by .define
- a label, as describe ahead
- an expression between parentheses, like (1+3*cte)

An expression can be:

- a value
- -expression
- ::expression (bit reversion)
- expression+expression
- expression-expression
- expression*expression
- expression/expression

Comments

The PIO assembler ignores text in a line after // or ;

It also ignores text between /* and */

Labels

A label is a special king of .define where the value is the current program instruction offset. It can be defined by

```
{label}:
```

or

```
PUBLIC {label}:
```

Selected SDK Functions

The SDK functions for interacting with the PIO are in the library hardware_pio.

There are a few parameters that you see in many functions:

- pio selects one of the two PIOs and should be pio0 or pio1
- sm selects one of the four state machines within a PIO and should be an integer (index) between 0 and 3.
- config is a pointer to a pio_sm_config structure that stores the configuration of a state machine. You should manipulate this structure with the SDK functions and not by accessing its members.

State Machine Allocation

The `hardware_pio` library maintains a simple (but multicore safe) control of state machine usage in each PIO.

```
int pio_claim_unused_sm (PIO pio, bool required)
```

This function will return the index of an unused state machine in a PIO. This is the preferred way to select a state machine, as it avoids conflicts that may result if you use a fixed index. You still have to select the PIO.

If `required` is false, the function will return -1 if all state machines are in use (*claimed*) in the PIO. If `required` is true and there is no free state machine, the function will panic (send an error message to `stdio` and halt).

```
void pio_sm_claim (PIO pio, uint sm)
```

This function marks a state machine as in use (*claimed*). Panics if the state machine was already claimed.

```
void pio_sm_unclaim (PIO pio, uint sm)
```

This function marks a state machine as not in use.

```
bool pio_sm_is_claimed (PIO pio, uint sm)
```

Returns true if state machine is claimed.

Program Control

```
uint pio_add_program (PIO pio, const pio_program_t *program)
```

Loads a PIO program. Will find a location (offset) in the instruction memory where there is enough space for the program, load the instructions and return the offset where the program was loaded. If something goes wrong (like not having enough space), the function panics (writes an error message to standard output and halts).

```
void pio_sm_init (PIO pio, uint sm, uint initial_pc, const pio_sm_config *config)
```

Resets and configures the state machine. The PC is initialized with `initial_pc` and the state machine is disabled (stopped).

```
static void pio_sm_set_config (PIO pio, uint sm, const pio_sm_config *config)
```

Configures a state machine (see configuration below to see how to prepare config)

```
static void pio_sm_set_enabled (PIO pio, uint sm, bool enabled)
```

This function will enable (start, `enabled = true`) or disable (stop, `enabled = false`) a state machine.

FIFO Usage

```
static void pio_sm_clear_fifos (PIO pio, uint sm)
```

Clear the FIFOs of the state machine sm (0 a 3) of the PIO pio.

```
static uint32_t pio_sm_get (PIO pio, uint sm)
```

Reads a word from the Rx FIFO of a state machine. Does not check if the FIFO is empty, the return is undefined if the FIFO is empty.

```
static uint32_t pio_sm_get_blocking (PIO pio, uint sm)
```

Reads a word from the Rx FIFO of a state machine, blocking (waiting in a loop for data) if it is empty.

```
static uint pio_sm_get_rx_fifo_level (PIO pio, uint sm)
```

Returns the number of words in the Rx FIFO of a state machine.

```
static uint pio_sm_get_tx_fifo_level (PIO pio, uint sm)
```

Returns the number of words in the Tx FIFO of a state machine.

```
static bool pio_sm_is_rx_fifo_empty (PIO pio, uint sm)
```

Return true if the Rx FIFO of a state machine is empty, false if holds data.

```
static bool pio_sm_is_rx_fifo_full (PIO pio, uint sm)
```

Return true if the Rx FIFO of a state machine is full, false if there is space for more data.

```
static bool pio_sm_is_tx_fifo_empty (PIO pio, uint sm)
```

Return true if the Tx FIFO of a state machine is empty, false if holds data.

```
static bool pio_sm_is_tx_fifo_full (PIO pio, uint sm)
```

Return true if the Tx FIFO of a state machine is full, false if there is space for more data.

```
static void pio_sm_put (PIO pio, uint sm, uint32_t data)
```

Writes a word in the Tx FIFO of a state machine. Does not check if the Tx FIFO is full, if it is the data is ignored.

```
static void pio_sm_put_blocking (PIO pio, uint sm, uint32_t data)
```

Writes a word to the Tx FIFO of a state machine, blocking (waiting in a loop for space) if it is full.

Configuration

```
static void pio_sm_set_clkdiv (PIO pio, uint sm, float div)
```

Sets the clock divisor for a state machine. The divisor is specified as a float number.

```
static void pio_sm_set_clkdiv_int_frac (PIO pio, uint sm, uint16_t div_int, uint8_t div_frac)
```

Sets the clock divisor for a state machine. The divisor is specified as an integer and a fraction (in units of 1/256). For example, `div_int=2` and `div_frac = 128` means 2.5.

```
static void pio_gpio_init (PIO pio, uint pin)
```

Connects a pin to a PIO. The documentation say this is needed for output pins only, put my third PIO example would not work without it... I recommend you use it for all pins, output or input.

```
void pio_sm_set_consecutive_pindirs (PIO pio, uint sm, uint pin_base, uint pin_count,
bool is_out)
```

Sets the direction of `pin_count` pins, starting from `pin_base`, in a state machine.

```
void pio_sm_set_pindirs_with_mask (PIO pio, uint sm, uint32_t pin_dirs, uint32_t pin_
mask)
```

Sets the direction of pins in a state machine. Bits with 1 in `pin_maks` indicate the pins that will be affected; the corresponding pin in `pin_dirs` define the direction (1 = output, 0 = input).

```
void pio_sm_set_pins (PIO pio, uint sm, uint32_t pin_values)
```

Sets the value of all pins in a state machine.

```
void pio_sm_set_pins_with_mask (PIO pio, uint sm, uint32_t pin_values, uint32_t pin_mask)
```

Sets the value of pins in a state machine. Bits with 1 in `pin_maks` indicate the pins that will be affected; the corresponding pin in `pin_values` define the value.

```
static pio_sm_config pio_get_default_sm_config (void)
```

Returns an initialized configuration structure. Configurations are set as following:

Configuration	Value
Out Pins	32 starting at 0
Set Pins	0 starting at 0
In Pins	(base) 0
Side Set Pins (base)	0
Side Set	disabled
Wrap	wrap=31, wrap_to=0
In Shift	shift_direction=right, autopush=false, push_thrshold=32
Out Shift	shift_direction=right, autopull=false, pull_thrshold=32
Jmp	Pin 0
Out Special	sticky=false, has_enable_pin=false, enable_pin_index=0
Mov Status	status_sel=STATUS_TX_LESSTHAN, n=0

Writes a word in the Tx FIFO of a state machine, blocking (waiting in a loop for space) if it is full.

```
static void sm_config_set_in_shift (pio_sm_config *c, bool shift_right, bool autopush,
uint push_threshold)
```

Sets the input shift register options (ISR) in a state machine configuration:

- If `shift_right` is true, the ISR will shift to right. If its false, the ISR will shift to the left.

- If `autopush` is true, the IRS will be pushed into the Rx FIFO when `push_threshold` bits are shifted in

```
static void sm_config_set_out_shift (pio_sm_config *c, bool shift_right, bool autopull,
uint pull_threshold)
```

Sets the output shift register options (OSR) in a state machine configuration:

- If `shift_right` is true, the OSR will shift to right. If its false, the OSR will shift to the left.
- If `autopull` is true, the ORS will be loaded from the Tx FIFO when `pull_threshold` bits are shifted out

```
static void sm_config_set_fifo_join (pio_sm_config *c, enum pio_fifo_join join)
```

Sets the join FIFO option in a state machine configuration. The values for join are:

- `PIO_FIFO_JOIN_NONE` use both Rx and Tx FIFOs, each with 4 positions
- `PIO_FIFO_JOIN_TX` use only a 8 position Tx FIFO
- `PIO_FIFO_JOIN_RX` use only a 8 position Rx FIFO

```
static void sm_config_set_out_pins (pio_sm_config *c, uint out_base, uint out_count)
```

Sets the “out” pins in a state machine configuration. `out_base` is the number of the first pin and `out_count` is the number of pins.

```
static void sm_config_set_set_pins (pio_sm_config *c, uint set_base, uint set_count)
```

Sets the “set” pins in a state machine configuration. `set_base` is the number of the first pin and `set_count` is the number of pins.

```
static void sm_config_set_in_pins (pio_sm_config *c, uint in_base)
```

Sets the “in” pins in a state machine configuration. `in_base` is the number of the first pin and `in_count` is the number of pins.

```
static void sm_config_set_sideset (pio_sm_config *c, uint bit_count, bool optional, bool
pindirs)
```

Sets the “side set” pins in a state machine configuration. `sideset_base` is the number of the first pin, `bit_count` is the number of pin. If `optional` is true, an additional bit in the side-set field will indicate if side-set is to be used for each instruction. If `pindirs` is true, side set will affect pin directions instead of values.

```
static void sm_config_set_wrap (pio_sm_config *c, uint wrap_target, uint wrap)
```

Configures program wrapping in a state machine configuration. When execution reaches the offset `wrap` the machine will jump to `wrap_target`.

```
static void sm_config_set_jmp_pin (pio_sm_config *c, uint pin)
```

Sets, in a state machine configuration, the pin tested by the JMP instruction to pin

```
static void sm_config_set_mov_status (pio_sm_config *c, enum pio_mov_status_type
status_sel, uint status_n)
```

Defines, in a state machine configuration, what will be the STATUS source in the MOV instruction. The options for status_sel are:

- STATUS_TX_LESSTHAN STATUS will be all-ones if TX FIFO level < status_n, otherwise all-zeros
- STATUS_RX_LESSTHAN STATUS will be all-ones if RX FIFO level < status_n, otherwise all-zeros

Miscellaneous Functions

```
static void pio_interrupt_clear (PIO pio, uint pio_interrupt_num)
```

Clears a PIO interrupt flag.

```
static bool pio_interrupt_get (PIO pio, uint pio_interrupt_num)
```

Returns true if the interrupt flag is set, false if it is cleared.

```
static void pio_sm_exec (PIO pio, uint sm, uint instr)
```

Execute the instruction instr in the State Machine sm (0 a 3) of PIO pio.

Examples

Here I am listing only the .pio file for each example. The associate CMakeLists.txt and .c file are in the github repository,

A simple square wave generator

Let's start simple.

The following program changes a pin between high and low. The pins is updated by SET PINS instructions. Auto-wrap will make the execution wrap back to the beginning when the last instruction is executed.

From where does the sqwave_program_get_default_config() function comes? It is created by the pio assembler and goes into the squarewave.pio.h include file. This function calls pio_get_default_sm_config() and changes the configurations related to the directives wrap, wrap_target and side_set.

Square Wave Generator

```

1 ;
2 ; Square Wave Generator - PIO Exemple for 'Knowing the RP2040' book
3 ; Copyright (c) 2022, Daniel Quadros
4 ;
5
6 .program sqwave
7
8 .wrap_target
9     set PINS, 1
10    set PINS, 0
11 .wrap
12
13 % c-sdk {
14 // Helper function to set a state machine to run our PIO program
15 static inline void sqwave_program_init(PIO pio, uint sm, uint offset, uint pin,
16     float freq) {
17
18     // Get an initialized config structure
19     pio_sm_config c = sqwave_program_get_default_config(offset);
20
21     // Map the state machine's SET pin group to one pin, namely the `pin`
22     // parameter to this function.
23     sm_config_set_set_pins(&c, pin, 1);
24
25     // Set this pin's GPIO function (connect PIO to the pad)
26     pio_gpio_init(pio, pin);
27
28     // Set the pin direction to output at the PIO
29     pio_sm_set_consecutive_pindirs(pio, sm, pin, 1, true);
30
31     // Configure the clock, the period of the square wave will two PIO cycles
32     float div = clock_get_hz(clk_sys) / (freq * 2);
33     sm_config_set_clkdiv(&c, div);
34
35     // Load our configuration, and jump to the start of the program
36     pio_sm_init(pio, sm, offset, &c);
37
38     // Set the state machine running
39     pio_sm_set_enabled(pio, sm, true);
40 }
41 %}

```

Sending data serially

In this example, the PIO gets a 12 bit value from the FIFO and shifts it out through a pin (LSB first). A pulse (clock) is generated in a second pin for each bit shifted.

The data pin is updated by an OUT PINS instruction and the clock pin is controlled by side set.

Much of the functionality comes from the configuration of the Tx FIFO. By setting `auto_pull`, the OSR will be automatically loaded from the FIFO, stalling if there is no data. The direction and number of bits to shift is also part of the configuration.

Serial Data Transmitter

```

1 ;
2 ; Serial data/clock transmitter - PIO Example for 'Knowing the RP2040' book
3 ; Copyright (c) 2022, Daniel Quadros
4 ;
5
6 .program serialtx
7 .side_set 1
8
9 .wrap_target
10     out pins,1 side 0
11     nop side 1
12 .wrap
13
14 % c-sdk {
15 // Helper function to set a state machine to run our PIO program
16 static inline void serialtx_program_init(PIO pio, uint sm, uint offset,
17     uint dataPin, uint clockPin, float freq) {
18
19     // Get an initialized config structure
20     pio_sm_config c = serialtx_program_get_default_config(offset);
21
22     // Map the state machine's OUT pin group to one pin, namely the `dataPin`
23     // parameter to this function.
24     sm_config_set_out_pins(&c, dataPin, 1);
25
26     // Map the state machine's SIDE SET pin group to one pin, namely the `clockPin`
27     // parameter to this function.
28     sm_config_set_sideset_pins(&c, clockPin);
29
30     // Set the pins GPIO function (connect PIO to the pad)
31     pio_gpio_init(pio, dataPin);
32     pio_gpio_init(pio, clockPin);

```

```

33
34 // Set the pins directions to output at the PIO
35 pio_sm_set_pindirs_with_mask(pio, sm, (1u << dataPin) | (1u << clockPin),
36     (1u << dataPin) | (1u << clockPin));
37
38 // Set the Tx FIFO
39 sm_config_set_out_shift (&c, true, true, 12);
40 sm_config_set_fifo_join(&c, PIO_FIFO_JOIN_TX);
41
42 // Configure the clock, the bit time will two PIO cycles
43 float div = clock_get_hz(clk_sys) / (freq * 2);
44 sm_config_set_clkdiv(&c, div);
45
46 // Load our configuration, and jump to the start of the program
47 pio_sm_init(pio, sm, offset, &c);
48
49 // Set the state machine running
50 pio_sm_set_enabled(pio, sm, true);
51 }
52 %}

```

Receiving clocked serial data

This example receives data sent by the previous example.

We will wait for the clock pin to change from 0 to 1, shift right the data pin into the ISR and wait for the clock pin to return to 0. For this to work, the clock frequency in the receiver must be greater than in the transmitter (at least 1.5x faster, as we will execute 3 instructions while the transmitter is executing 2).

As the ISR is 32 bits, we are shifting to the right and each received value is 16 bits, the result will be in the upper 16 bits of the words in the FIFO. This is a common issue when doing serial communication with the PIO. One solution is to use an ARM core to shift data before transmitting or after receiving. If we are shifting a multiple of 8, we can bypass the SDK functions and access directly the FIFO as bytes or 16 bit words.

To simplify things, I am imposing that the data pin and clock pin are consecutive. They are mapped in the IN group so PINS 0 is the data pin and PINS 1 is the clock pin.

Serial Data Receiver

```

1  ;
2  ; Serial data/clock receiver - PIO Example for 'Knowing the RP2040' book
3  ; Copyright (c) 2022, Daniel Quadros
4  ;
5
6  .program serialrx
7
8  .wrap_target
9      wait 1 pin 1      // wait for clock high
10     in pins, 1        // shift in data
11     wait 0 pin 1      // wait for clock back to low
12 .wrap
13
14 % c-sdk {
15 // Helper function to set a state machine to run our PIO program
16 static inline void serialrx_program_init(PIO pio, uint sm, uint offset,
17     uint dataPin, float freq) {
18
19     // Get an initialized config structure
20     pio_sm_config c = serialrx_program_get_default_config(offset);
21
22     // Map the state machine's IN pin group to pins starting at `dataPin`
23     sm_config_set_in_pins(&c, dataPin);
24
25     // Set the pins GPIO function (connect PIO to the pad)
26     pio_gpio_init(pio, dataPin);
27     pio_gpio_init(pio, dataPin+1);
28
29     // Set the pins directions to input at the PIO
30     pio_sm_set_pindirs_with_mask(pio, sm, (3u << dataPin), 0);
31
32     // Configure the Rx FIFO
33     sm_config_set_in_shift (&c, true, true, 12);
34     sm_config_set_fifo_join(&c, PIO_FIFO_JOIN_RX);
35
36     // Configure the clock, we will use double of the transmitter's clock
37     float div = clock_get_hz(clk_sys) / (freq * 4);
38     sm_config_set_clkdiv(&c, div);
39
40     // Load our configuration, and jump to the start of the program
41     pio_sm_init(pio, sm, offset, &c);
42

```



```

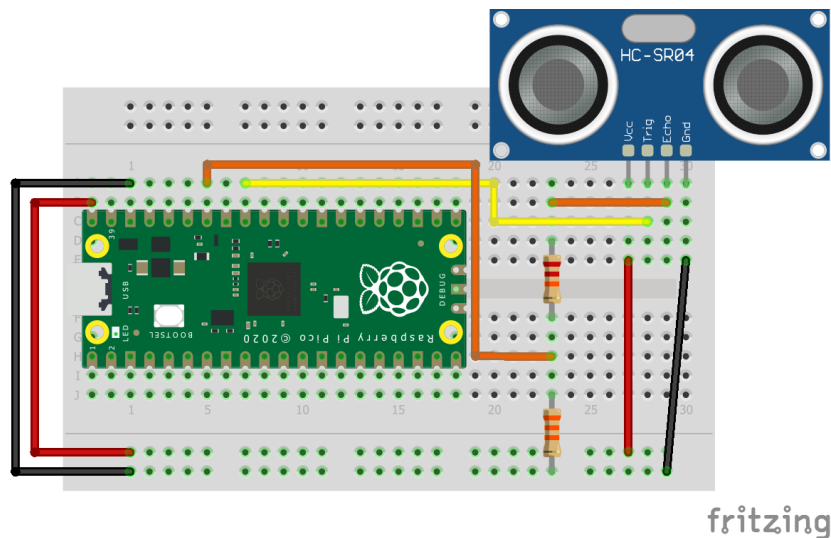
43     // Set the state machine running
44     pio_sm_set_enabled(pio, sm, true);
45 }
46 %}

```

Interfacing a HC-SR04 Ultrasonic Sensor

The HC-SR04 is a popular distance sensor, with two pins: *trigger* (input) and *echo* (output). It will send an ultrasonic signal when a 10 ms pulse is given to the trigger pin. After that, the echo pin will stay high until an echo is detected or 38 ms has passed. This sensor works for (at least) distances between 2.5 cm (150 μ s echo) and 4,3 m (25 ms echo).

When wiring this sensor to the RP2040, care must be taken to power it with 5V (I could not get reliable results if 3.3V) and to add some kind of level conversion for Echo signal. In my tests I used a resistive divisor:



Connecting the HC-SR04 sensor to the Pi Pico

The first decision is what clock to use in the PIO. As we want to measure (with a good precision) a time between 150 μ s and 38ms, a half microsecond cycle (2MHz clock) is a good choice.

The PIO program is not complicated, but there are some PIO instructions peculiarities that need to be addressed:

- A register can only be initialized (through the SET instruction) with a value between 0 and 31. The initial counter value is gotten from the Tc FIFO and moved to the counting register.
- The WAIT instruction has no timeout. If no sensor is connected the program will stall waiting for echo to go high.
- The JMP instruction can only JMP on a high in a pin. As we are testing for a low, the code is a little convoluted.

- When using JMP for a loop, it will jump if the counter is not zero and then decrement. If the counter is zero it will not jump. A little unusual, but works fine in our case.

The PIO program bellow receives (through the Tx FIFO) the timeout value (I used 150ms) and sends back (through the Rx FIFO) the remaining counter. From this we can calculate the number of microseconds for echo to go down. The distance can be found remembering that the signal had to go and return and the speed of sound is 343m/s (see the main program in github).

HC-SR04 Interface

```

1  ;
2  ; Interface to HC-SR04 sensor - PIO Example for 'Knowing the RP2040' book
3  ; Copyright (c) 2022, Daniel Quadros
4  ;
5
6  .program hcsr04
7
8  .wrap_target
9      // wait for a request
10     pull
11     mov x, osr      // data is timeout
12
13     // generate a 10 usec (20 cycles) trigger pulse
14     set pins, 1 [19]
15     set pins, 0
16
17     // wait for the start of the echo pulse
18     wait 1 pin 0
19
20     // wait for the end of the echo pulse
21     // decrements x each 2 cycles (1 usec)
22 wait_for_echo:
23     jmp pin, continue
24     jmp done
25 continue:
26     jmp x--, wait_for_echo
27 done:
28     mov isr, x
29     push
30 .wrap
31
32 % c-sdk {
33 // Helper function to set a state machine to run our PIO program
34 static inline void hcsr04_program_init(PIO pio, uint sm, uint offset,
```

```
35     uint triggerPin, uint echoPin) {
36
37     // Get an initialized config structure
38     pio_sm_config c = hcsr04_program_get_default_config(offset);
39
40     // Map the state machine's pin groups
41     sm_config_set_set_pins(&c, triggerPin, 1);
42     sm_config_set_in_pins(&c, echoPin);
43     sm_config_set_jump_pin(&c, echoPin);
44
45     // Set the pins directions at the PIO
46     pio_sm_set_consecutive_pindirs(pio, sm, triggerPin, 1, true);
47     pio_sm_set_consecutive_pindirs(pio, sm, echoPin, 1, false);
48
49     // Make sure trigger is low
50     pio_sm_set_pins_with_mask(pio, sm, 1 << triggerPin, 0);
51
52     // Set the pins GPIO function (connect PIO to the pad),
53     pio_gpio_init(pio, triggerPin);
54     pio_gpio_init(pio, echoPin);
55
56     // Configure the FIFOs
57     sm_config_set_in_shift (&c, true, false, 1);
58     sm_config_set_out_shift (&c, true, false, 1);
59
60     // Configure the clock for 2 MHz
61     float div = clock_get_hz(clk_sys) / 2000000;
62     sm_config_set_clkdiv(&c, div);
63
64     // Load our configuration, and jump to the start of the program
65     pio_sm_init(pio, sm, offset, &c);
66
67     // Set the state machine running
68     pio_sm_set_enabled(pio, sm, true);
69 }
70 %}
```

Communication Using I²C

I²C is a very popular electrical protocol for connecting all kind of devices to microcontrollers.

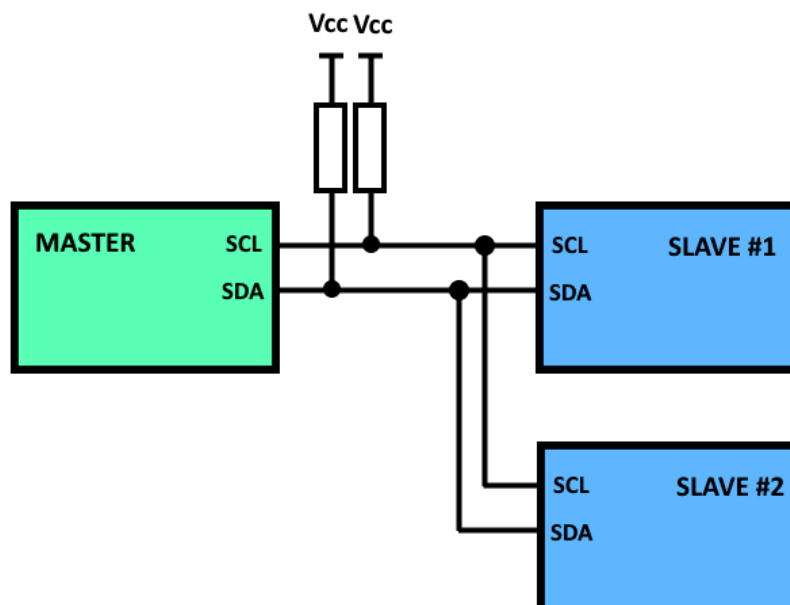
I²C Basics

The objective of I²C is to allow the simple short distance connection of multiple low-to-medium speed devices. While it was initially envisioned for in-board connections between integrated circuits (hence the name *Inter-Integrated Circuit*), today you can find many modules (like sensors, displays and real-time clocks) that use it.

I²C Topology

I²C is organized as a multiple drop bus, using only two connection (“wires”).

I²C distinguishes between **masters** (or *controllers*) and **slaves** (or *targets*). I will use the original terminology (master/slave) as this is what you will find in most of the literature.



I²C Topology

Each slave should have a unique **address**. All communications are started by a master, that selects a slave by its address and inform if it should receive or transmit.

In this text I will concentrate in the most common configuration where there is only one master and the slaves have 7-bit addresses.

7-bit I²C addresses in the binary form of 0000xxx and 1111xxx are reserved for special functions and should not be used by slaves.

Electrical Interface

The two signals used by I²C are:

- SCL: this is the clock that marks where are the bits. This line is always driven by the master.
- SDA: this is the data line and is bi-directional.

To allow the direct connection of multiple devices in a single wire, the devices must have:

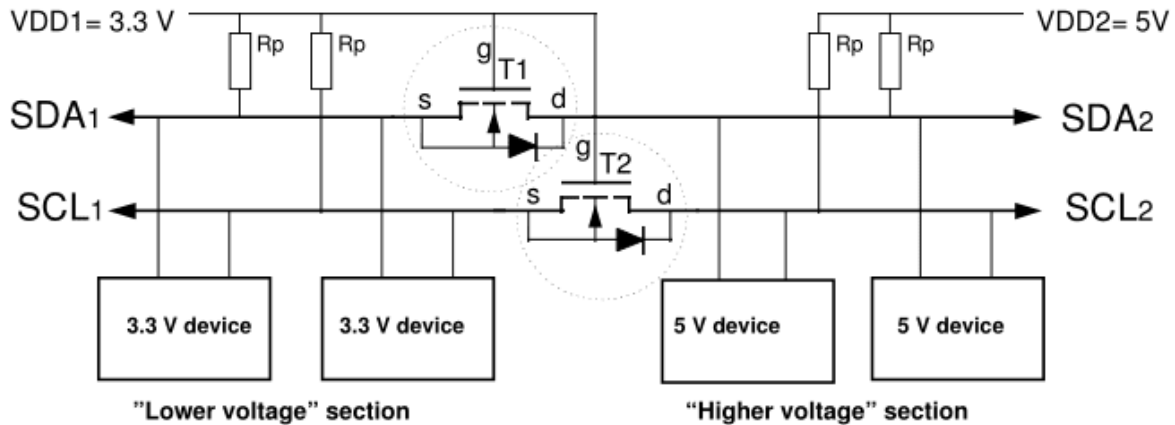
- An input buffer to read the level in the wire.
- An open drive driver that can pull the wire to low level or allow it to fluctuate.

Pull-up resistors guarantee that the signal is high if no device is pulling it to low. The value of this resistors depends upon:

- The capacitance of the connection. Higher capacitance requires lower resistors to guarantee that the signals will change in short time.
- The speed of the communication. Higher speeds require faster signals change which requires lower resistors
- Allowed power consumption. Lower resistors will consume more power.

In some cases the pull-up resistors in the RP2040 PAD will be enough. Also many modules contain pull-up resistors.

Connecting 3.3V (like the RP2040) and 5V devices directly in the same I²C bus is not recommended, as the 3.3V device will be submitted to voltages slightly above 3.3V. Nevertheless, it is common practice for hobbyists. For professional designs you should use an I²C level converter or use MOSFETs as in the circuit bellow.

Interconnection of 3.3V and 5V I²C devices

Some clock speeds (*modes*) are standardized:

- *standard*: 100 kbits/s
- *fast*: 400 kbit/s
- *fast plus*: 1 Mbit/s
- *high-speed*: 3.4 Mbit/s
- *ultra-fast*: 5 Mbit/s

The RP2040 does not support these last two modes.

Start and Stop Conditions

The idle condition is for the two signals be at HIGH level (due to the pull-up resistors, as no one is pulling the signals down).

During normal communication the transmitter should only change SDA when SCL is LOW. The receiver will read SDA when SCL changes from LOW to HIGH.

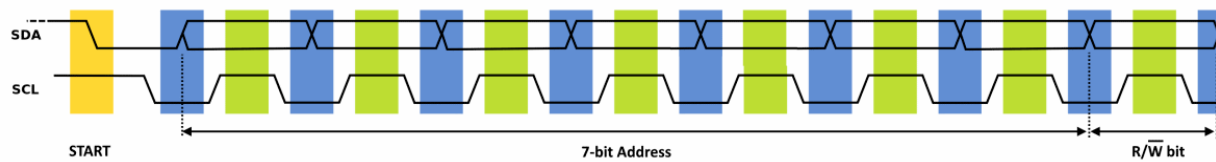
There are two special conditions that violate this rule, to signal the start and end of a **transaction**:

- A **Start** condition is generated by pulling SDA LOW while SCL is HIGH. This marks the start of a transaction.
- A **Stop** condition is generated by pulling SDA LOW while SCL is LOW, letting SCL go HIGH and then letting SDA go HIGH. This marks the end of a transaction.

The Stop can be combined with a Start to start a new transaction without release the bus: let SDA go HIGH then let SCL go HIGH and then pull SDA LOW. This is called a **Restart**.

These conditions are always generated by the master.

After a START, the master will send the address of the slave and signal if its a read or write operation:



I²C Start Condition and Device Addressing

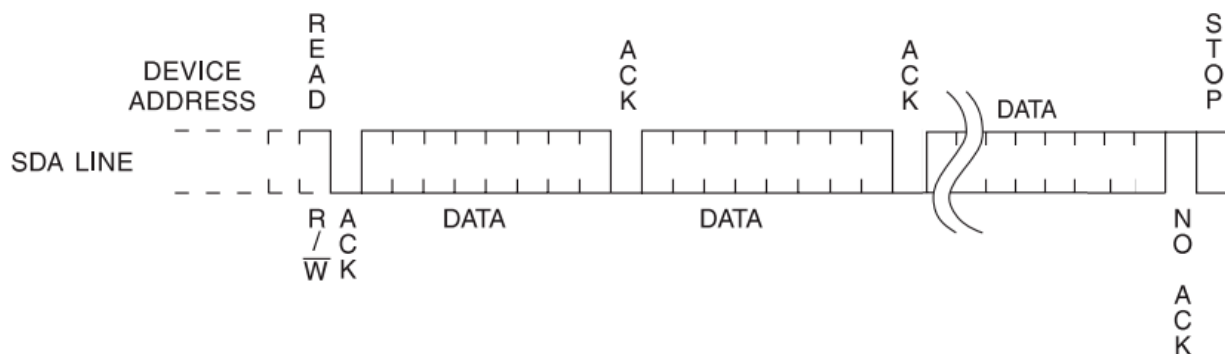
In the figure above, the blue areas indicate where the master sets the SDA line and the green areas indicate where the slave reads the SDA line. The first 7 bits after the start is the slave address, the last bit is 0 for a write and 1 for a read.

The slave addressed must acknowledge the selection by sending a “0”.

Read Operation

A read operation follows these steps:

- SCL and SDA are high (idle).
- The master pulls down SDA, signaling a start condition. After that the master will pulse SCL for each bit, the transmitter will change SDA when SCL is LOW and the receiver will read SDA when SCL changes to HIGH.
- The master sends the slave address, followed by a “1” bit (indicating read).
- The slave pulls down SDA, acknowledging the address.
- The slave controls SDA, sends 8 bits and releases SDA.
- The master keeps SDA HIGH for 1 bit to request another byte or pull it LOW (followed by a STOP condition) to end the transaction.

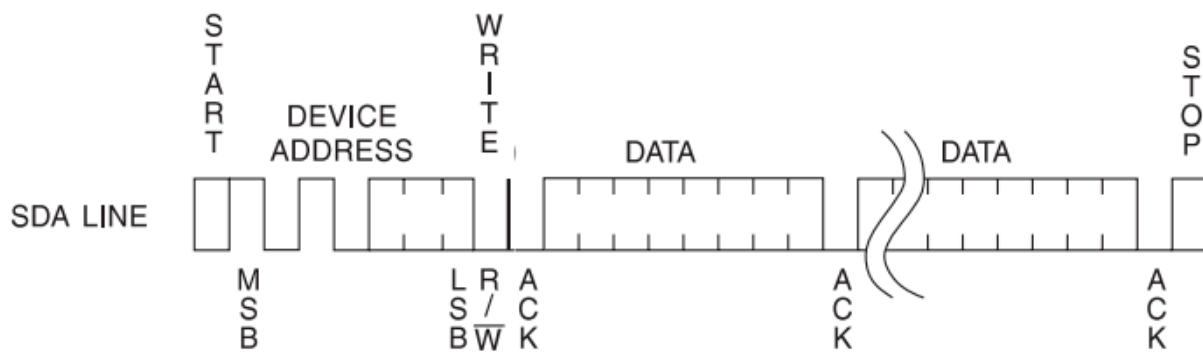


I²C Read Operation

Write Operation

A write operation follows these steps:

- SCL and SDA are high (idle).
- The master pulls down SDA, signaling a start condition. After that the master will pulse SCL for each bit, the transmitter will change SDA when SCL is LOW and the receiver will read SDA when SCL changes to HIGH.
- The master sends the slave address, followed by a “0” bit (indicating write).
- The slave pulls down SDA, acknowledging the address.
- The master controls SDA, sends 8 bits and releases SDA.
- The slave pulls down SDA, acknowledging the data.
- The previous two steps can be repeated for more bytes
- The master sends a STOP condition to signal the end of the transaction



I²C Write Operation

Combined Write/Read Operation

Many devices use some kind of internal addressing, such as a memory address or a register address. This address is normally sent by the master as the first bytes written. As a result, a read operation on a device is actually an I²C write transaction followed by an I²C read transaction. This is a good opportunity to combine the STOP and START condition.

In the examples we will see this when interfacing a EEPROM.

I²C in the RP2040

The RP2040 has two I²C peripherals with the following features:

- Can be used as master or slave.
- Support for standard, fast and fast plus mode.
- Supports 10-bit address in master mode
- 16 position transmit and receive FIFOs
- Can generate interrupts and work with DMA

Each position in the FIFOs can hold not only the byte to transmit, but also flags. In the receive FIFO, a flag signals if the data is the first byte received after the address. In the transmit FIFO, flags select read or write operation and control restart and stop condition generation. As a result of this controls been together with the data, a zero length operation is not supported.

Clock Generation

In master mode, clock is generate from `clk_sys` by the I²C peripheral. The hardware gives fine control over the clock with the following configurations:

- mode (standard, fast or fast plus)
- low, high and minimum data setup times

The SDK functions programs the appropriate values from the baud rate.

Pins Options

he RP2040 has a somewhat flexible mapping of pins for the serial interfaces (UART, SPI and I2C).

The options for I2C0 are:

Function	GPIOs
SDA	0, 4, 8, 12, 16, 20, 24, 28
SCL	1, 5, 9, 13, 17, 21, 25, 29

The options for I2C1 are:

Function	GPIOs
SDA	2, 6, 10, 14, 18, 22, 26
SCL	3, 7, 11, 15, 19, 23, 27

Selected SDK Functions

The I²C functions are in the library `hardware_i2c`. The `i2c` parameter should be `i2c0` or `i2c1`.

```
uint i2c_init (i2c_inst_t *i2c, uint baudrate)
```

Initializes a I²C peripheral for master mode, setting the clock configurations for baudrate. For slave mode, call `i2c_set_slave_mode` after this function; baudrate must be informed (although clock is not generated in this case) for right configuration.

This function must be called before the others.

Returns the actual baudrate.

```
void i2c_set_slave_mode (i2c_inst_t *i2c, bool slave, uint8_t addr)
```

Changes mode between master (`slave = false`) and slave (`slave = true`). In slave mode `addr` is the slave address.

```
static void i2c_write_raw_blocking (i2c_inst_t *i2c, const uint8_t *src, size_t len)
```

This routine will put into the transmit FIFO `len` bytes starting at `src`, waiting for space available at the FIFO.

This function is mainly for slave-mode operation; the bytes at the FIFO will be sent as requested by the master.

```
int i2c_write_blocking (i2c_inst_t *i2c, uint8_t addr, const uint8_t *src, size_t len, bool nostop)
```

```
static int i2c_write_timeout_us (i2c_inst_t *i2c, uint8_t addr, const uint8_t *src, size_t len, bool nostop, uint timeout_us)
```

```
int i2c_write_blocking_until (i2c_inst_t *i2c, uint8_t addr, const uint8_t *src, size_t len, bool nostop, absolute_time_t until)
```

These routines are mainly for master-mode operation and will try to send `len` bytes starting from `src`. `addr` is the slave address (a valid address must be given even in slave mode).

If `nostop` is false, a STOP condition will be generated after the last byte. In master mode a START condition will initiate the next transfer.

If `nostop` is true, the STOP condition will not be generated and a RESTART will be used at the start of the next transfer.

The blocking version will block indefinitely until all bytes are transfered (or the address is not acknowledged).

The `timeout_us` version allows to specify a timeout, in microseconds, for the entire transaction to complete.

In the `until` version the timeout is specified by a maximum finish time.

Returns the number of bytes sent or `PICO_ERROR_TIMEOUT`(in case of a timeout) or `PICO_ERROR_GENERIC` (if something else went wrong, like the device not acknowledging the address).

```
static void i2c_read_raw_blocking (i2c_inst_t *i2c, uint8_t *dst, size_t len)
```

This routine will put `len` bytes from the receive FIFO into the memory starting at `dst`, waiting for data available at the FIFO.

This function is mainly for slave-mode operation.

```
int i2c_read_blocking (i2c_inst_t *i2c, uint8_t addr, uint8_t *dst, size_t len, bool
nostop)
```

```
static int i2c_read_timeout_us (i2c_inst_t *i2c, uint8_t addr, uint8_t *dst, size_t len,
bool nostop, uint timeout_us)
```

```
int i2c_read_blocking_until (i2c_inst_t *i2c, uint8_t addr, uint8_t *dst, size_t len,
bool nostop, absolute_time_t until)
```

These routines are mainly for master-mode operation and will try to read `len` bytes to the memory starting at `dst.addr` is the slave address (a valid address must be given even in slave mode).

If `nonstop` is false, a STOP condition will be generated after the last byte. In master mode a START condition will initiate the next transfer.

If `nonstop` is true, the STOP condition will not be generated and a RESTART will be used at the start of the next transfer.

The blocking version will block indefinitely until all bytes are transfered (or the address is not acknowledged).

The `timeout_us` version allows to specify a timeout, in microseconds, for the entire transaction to complete.

In the until version the timeout is specified by a maximum finish time.

Returns the number of bytes read or `PICO_ERROR_TIMEOUT`(in case of a timeout) or `PICO_ERROR_GENERIC` (if something else went wrong, like the device not acknowledging the address).

```
static size_t i2c_get_write_available (i2c_inst_t *i2c)
```

Returns the number of empty positions in the transmit FIFO (the number of bytes that can be written without blocking).

```
static size_t i2c_get_read_available (i2c_inst_t *i2c)
```

Returns the number of filled positions in the receive FIFO (the number of bytes that can be read without blocking).

```
static uint i2c_get_dreq (i2c_inst_t *i2c, bool is_tx)
```

Returns the DREQ to use for transferring data via DMA to/from the I²C peripheral. `is_tx` specifies the direction (true = transfer data to transmit, false = transfer received data).

Examples

I²C Scanner

The fact that a slave must acknowledge its address allows do find the addresses of the devices connected to a master. We just need to do a dummy 1 byte read at all 127 address (except the

reserved ones) and check which are acknowledged. This is what this example does.

I²C Scanner

```
1  /**
2   * @file i2cscanner.c
3   * @author Daniel Quadros
4   * @brief Finding out the addresses of connected I2C devices
5   * @version 0.1
6   * @date 2022-07-28
7   *
8   * @copyright Copyright (c) 2022, Daniel Quadros
9   *
10  */
11
12  #include "stdio.h"
13  #include "pico/stdlib.h"
14  #include "hardware/i2c.h"
15
16  // Select I2C and Pins
17  #define I2C_ID      i2c0
18  #define I2C_SCL_PIN  17
19  #define I2C_SDA_PIN  16
20
21  // I2C Configuration
22  #define BAUD_RATE 100000 // standard 100KHz
23
24  // Main Program
25  int main() {
26      // Start stdio and wait for USB connection
27      stdio_init_all();
28      while (!stdio_usb_connected()) {
29          sleep_ms(100);
30      }
31
32      // Set up I2C
33      uint baud = i2c_init (I2C_ID, BAUD_RATE);
34      printf ("I2C @ %u Hz\n", baud);
35
36      // Set up the I2C pins
37      gpio_set_function(I2C_SCL_PIN, GPIO_FUNC_I2C);
38      gpio_set_function(I2C_SDA_PIN, GPIO_FUNC_I2C);
39      gpio_pull_up(I2C_SCL_PIN);
40      gpio_pull_up(I2C_SDA_PIN);
```

```

41
42
43     printf("Scanning I2C devices...\n");
44     printf("    0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F\n");
45
46     for (int addr = 0; addr <= 0x7F; ++addr) {
47         if ((addr % 16) == 0) {
48             printf("%02x ", addr);
49         }
50
51         // scan only non-reserved address
52         int ret = PICO_ERROR_GENERIC;
53         if (((addr & 0x78) != 0) && ((addr & 0x78) != 0x78)) {
54             uint8_t rxdata;
55             ret = i2c_read_blocking(i2c_default, addr, &rxdata, 1, false);
56         }
57         printf(ret < 0 ? "." : "X");
58         printf((addr % 16) == 15 ? "\n" : " ");
59     }
60     printf("Done.\n");
61
62     // Main loop
63     while (1) {
64         sleep_ms(1000);
65     }
66 }
67

```

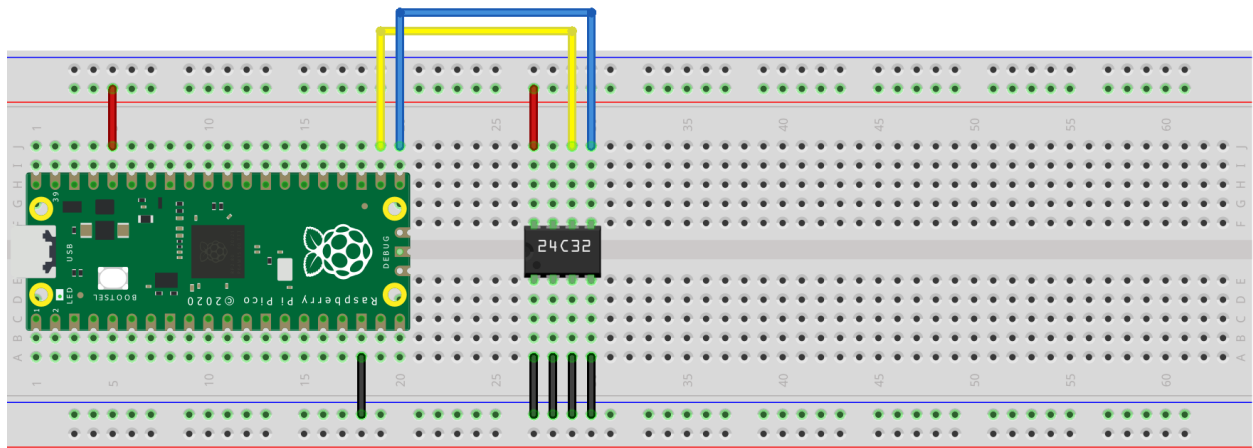
You can test this example with the next example circuit.

Using a 24C32 EEPROM

The 24C IC series features *Electrical Erasable Programmable Read-Only Memories (EEPROM)* of various sizes with I²C interface. EEPROM are non-volatile memories (that is, they retain their data when not powered) that can be reprogrammed in-circuit (so not really “read-only”). While EEPROM can be read as many times as needed, writing (or, more precisely, erasing) can be done only a number of time and takes much more time than reading.

EEPROMs are good for storing configuration and all kind of data that needs to be kept even if the circuit is powered down.

In this example we will use a 24C32 that has 32 kbits (organizes as 4 kbytes). The datasheet for Atmel’s AT24C32 specifies data retention of 100 years, 1 million write cycles and a write time of 10 ms maximum. The I²C interface can operate at up to 100kHz at 3.3V (and up to 400kHz at 5V).



fritzing

Connecting the 24C32 to the Raspberry Pi Pico

Accessing the 24C32

```

1  /**
2   * @file i2c24c32.c
3   * @author Daniel Quadros
4   * @brief Accessing a 24C32 EEPROM using I2C
5   * @version 0.1
6   * @date 2022-07-28
7   *
8   * @copyright Copyright (c) 2022, Daniel Quadros
9   *
10  */
11
12  #include "stdio.h"
13  #include "pico/stdlib.h"
14  #include "hardware/i2c.h"
15
16  // Select I2C and Pins
17  #define I2C_ID      i2c0
18  #define I2C_SCL_PIN  17
19  #define I2C_SDA_PIN  16
20
21  // I2C Configuration
22  #define BAUD_RATE 100000 // standard 100KHz
23
24  // EEPROM
25  #define EEPROM_ADDR 0x50
26  #define PAGE_SIZE   32
27

```

```

28 // Main Program
29 int main() {
30     // Start stdio and wait for USB connection
31     stdio_init_all();
32     while (!stdio_usb_connected()) {
33         sleep_ms(100);
34     }
35
36     // Set up I2C
37     uint baud = i2c_init (I2C_ID, BAUD_RATE);
38     printf ("I2C @ %u Hz\n", baud);
39
40     // Set up the I2C pins
41     gpio_set_function(I2C_SCL_PIN, GPIO_FUNC_I2C);
42     gpio_set_function(I2C_SDA_PIN, GPIO_FUNC_I2C);
43     gpio_pull_up(I2C_SCL_PIN);
44     gpio_pull_up(I2C_SDA_PIN);
45
46
47     printf("I2C Example: 24C32 EEPROM\n");
48
49     // Fill the first 256 bytes with 0x00 to 0xFF, using Page Write
50     uint8_t value = 0;
51     uint8_t buffer[PAGE_SIZE+2];
52     for (uint16_t addr = 0; addr < 0xFF; addr += PAGE_SIZE) {
53         // Write a page
54         printf ("\rWriting at 0x%02X", addr);
55         buffer[0] = addr >> 8;
56         buffer[1] = addr & 0xFF;
57         for (int i = 0; i < PAGE_SIZE; i++) {
58             buffer[i+2] = value++;
59         }
60         int ret = i2c_write_blocking (I2C_ID, EEPROM_ADDR, buffer, PAGE_SIZE+2, false\
61 e);
62         if (ret == (PAGE_SIZE+2)) {
63             // Wait for write to complete
64             // 24C32 will acknowledge address only when writting finished
65             while (i2c_read_blocking(I2C_ID, EEPROM_ADDR, buffer, 1, false) != 1) {
66                 sleep_ms(1);
67             }
68         } else {
69             printf ("*** Something went wrong ***\n");
70         }

```

```

71     }
72     printf ("\rWriting concluded.\n");
73
74     // Dump the first 256 bytes using sequential read
75     printf ("Reading EEPROM:\n");
76     uint8_t bufferRx[16];
77     for (uint16_t addr = 0; addr < 0xFF; addr += 16) {
78         buffer[0] = addr >> 8;
79         buffer[1] = addr & 0xFF;
80         int ret = i2c_write_blocking (I2C_ID, EEPROM_ADDR, buffer, 2, true);
81         if (ret == 2) {
82             ret = i2c_read_blocking(I2C_ID, EEPROM_ADDR, bufferRx, 16, false);
83             if (ret == 16) {
84                 printf ("0x%02X:", addr);
85                 for (int i = 0; i < 16; i++) {
86                     printf (" %02X", bufferRx[i]);
87                 }
88                 printf ("\n");
89             }
90         }
91     }
92     printf("Done.\n");
93
94     // Main loop
95     while (1) {
96         sleep_ms(1000);
97     }
98 }

```

A few points about the code:

- Multiple bytes can be written using the *page write*. This writes must stay inside 32 byte pages.
- The EEPROM will ignore all operations while a write is in progress. We can check the end of the writing by trying to address the 24C32, it will acknowledge only when its ready for another operation.
- Multiple bytes can be read using *sequential read*. This reads are not limited by the write pages.
- To do a memory read, first the initial address is written and then the bytes are read.

Asynchronous Serial Communication: the UARTs

Asynchronous serial communication is one of the oldest form of serial communication. Bits are sent serially (one after the other) over a wire with no common clock signal to synchronize the receiver to the transmitter and determine where are the individual bits. A communication speed (called, not precisely, *baud rate*) must be previously agreed by the two sides.

The RP2040 has two UARTs, with the following features:

- 32 position queues (*FIFOs* - First In First Out) for transmission and reception
- programmable baud rate generator
- support for 5, 6, 7 and 8 bits of data, 1 or 2 stop bits, parity none, even or odd (see framing in the next section)
- break detection and generation
- support for hardware flow control
- interrupt and DMA support

The UARTs in the RP2040 are based on the PL011 (a standard UART design by ARM), but does not implements all its features.

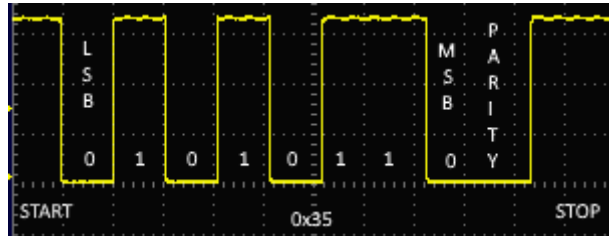
Framing

When no communication is taking place, the signal stays at a high (“1”) level.

Individual words (typically a byte) are sent as a “frame” composed of:

- **start bit:** The signal goes to low level and stays there for a “bit time” (as defined by the baud rate). The change from high to low signals the receiver that a frame is starting and is used to determine where individual bits will be.
- **data bits:** The individual bits of the word. The least significant bit is send first and the most significant bit is send last.
- **parity bit:** optional bit used to detect communication errors. If using *even parity*, the total number of “1” bits (considering the data bits and the parity bit) is even. In *odd parity* the total number of “1” bits is odd.

- **stop bit(s):** the signal is kept at high level to signal the end of the frame. Some really old equipment required two stop bits (that is, that the signal was kept high for at list two bit times before the next start bit), but nowadays one stop bit is standard. As the beginning of the next start bit is asynchronous to the stop bits, the line can be keep high for anytime after the minimum stop bit.



UART Framing - Transmission of 0x35 7 bits, even parity

A special condition (called *break*) is signaled by keeping the signal at low level for a whole frame (or more).

FIFOs

Each UART has two FIFOs, one for reception and one for transmission.

The transmission FIFO can store up to 32 8-bit words. Data written to this FIFO will be consumed by the transmitter. The Tx FIFO can be disabled to act like a single data-to-transmit register.

The reception FIFO holds 32 12-bit words. The Rx FIFO can be disabled to act like a single data-received register. The received data is in the lower 8 bits, the upper 4 bits contains the following flags:

- bit 11: **OE** (Overrun Error). This bit will be one if data is received when the FIFO is full, indicating that data was lost. This bit is not associated with the received data, it will only return to zero when a new frame is received and there is space for it in the FIFO.
- bit 10: **BE** (Break Error). If a break condition is detected, a word with this bit set and data equals zero will be put in the FIFO. A new word will be generated only after the line goes back to high level (ending the break condition).
- bit 9: **PE** (Parity Error). This bit will be one if data is received with the wrong parity;
- bit 8: **FE** (Framing Error). This bit will be one if a valid stop bit is not detected. This can be caused by a communication error, wrong frame format or wrong baud rate.

Control Signals and Hardware Flow Control

PC users may recall the use of serial communications with modems and phone lines to connect to the Internet. Standards define a number of control signals between a computer (or *DTE* - Data Transmission Equipment) and a modem (or *DCE* - Data Communication Equipment).

While the registers for the UARTs refer to many of these signals (mainly for compatibility with the chips used in old IBM PCs), the RP2040 actually supports only two of them:

- **RTS** (Request To Send) this is an output signal that goes (in the standard) to high level to indicate that the DTE wants to transmit
- **CTS** (Clear To Send) this is an input signal, high level means that the DCE can accept data from the DTE

The use of these signals is called **hardware flow control** as opposed to **software flow control** (where special characters or messages in the data signal when transmission must stop).

The RP2040 UARTs support the use of RTS and CTS in a non-standard way to control reception and/or transmission:

- In *RTS flow control*, the RTS signal is used to inform the other side when to transmit. It will be high as long as there is a configurable space in the reception queue. When the reception queue fills up, RTS goes down to inform the other side to stop transmission.
- In *CTS flow control*, transmission of each word only starts when CTS is high.

To use the hardware flow control, RTS and CTS pins must be configured and flow control enabled.

In a typical hardware flow control configuration, you will enable both options and cross the RTS and CTS signals of the two sides.

Baud Rate Generation

Strictly speaking, baud rate is the number of *symbols* transmitted per unit of time while bits per second (bps) is the number of bits transmitted per second. In the UART a symbol is one bit, so it is common to use the term baud rate when bits per second would be more appropriate.

The UART will generate the baud rate from its clock `clk_peri` (FUARTCLK in the docs) using a fractional divider, as long as

- `clk_peri` is at least 16 times the baud rate
- `clk_peri` is at most 16×65535 times the baud rate
- `clk_peri` is at most $5/3$ of the processor clock `clk_sys` (FPCLK in the docs)

The baud rate divisor is a 22-bit number consisting of a 16-bit integer and a 6-bit fractional part. It divides the UART clock to generate an internal Baud16 clock that is 16 times the baud rate.

For example, suppose we are using the standard 125MHz for `clk_sys` and `clk_peri`. To generate 9600 bps we need to use a divisor of $125000000/(16 \times 9600) = 813.802$. *813 is the integer part, the 6-bit fractional part is integer $(0.80264 \times 0.5) = 51$.*

Note that the resulting baud is not exact, as $51/64 = 0.796875$, The actual baud rate (not taking in account the clock precision) is $125000000/(16 \times 813.796875) = 9600.06$ (a very low error).

These calculations can be done by the `uart_set_baudrate()` function in the SDK.

UART Status and Interrupts

In the RP2040, the UART uses a single combined interrupt in the processors (UARTIMTR). There are eleven possible reasons for this interrupt and they can be independently masked:

- **UARTMSINTR:** modem status interrupt, generated when a modem status changes (in the RP2040 this can only be the CTS signal, the PL011 has provision for DCD, DSR and RI, totaling four possible sources). It is cleared by writing a 1 to the corresponding bit(s) in the Interrupt Clear Register (UARTICR).
- **UARTRXINTR:** receive interrupt. When the FIFO is enabled, this interrupt will be asserted when the FIFO reaches a programmable level. It will be cleared when the FIFO level drops below another programmable level (by reading the received data) or by writing a 1 to the corresponding bit(s) in the Interrupt Clear Register. If the FIFO is not enabled, both levels are one (that is, the interrupt will be assert when a byte is received and cleared when this byte is read - or the interrupt is cleared in the UARTICR).
- **UARTTXINTR:** transmit interrupt. When the FIFO is enabled, this interrupt will be asserted when the FIFO level is equal or lower a programmable level. It will be cleared when the FIFO level get above another programmable level (by transmitting the data in the FIFO) or by writing a 1 to the corresponding bit(s) in the Interrupt Clear Register. If the FIFO is not enabled, both levels are one (that is, the interrupt will be assert when the transmitter is free and cleared when the transmitter is busy - or the interrupt is cleared in the UARTICR).
- **UARTRTINTR:** receive timeout interrupt. This interrupt is assert when no data is received in 32 bit time and there is data in the FIFO. It is cleared when the FIFO is emptied (by reading the data) or by writing a 1 to the corresponding bit in the Interrupt Clear Register (UARTICR). This interrupt is normally used when the FIFO level for UARTRXINTR is greater than one, so data is not “forgotten” in the FIFO.
- **UARTEINTR:** error interrupt. Here we have the four errors we seem before: framing, parity, break detect and overrun. It can be cleared by writing to the relevant bits of the Interrupt Clear Register.

The programmable levels in the FIFO for the transmit and receive exists to reduce the number (and overhead) of interrupts.

Using the transmit interrupt requires a little logic:

- Start with the transmit interrupt disabled
- When you have something for transmission, first check if you can just put it in the UART FIFO. If so, there is no need for an interrupt. If the UART FIFO is full, you store the data in a FIFO of your own and enable the transmit interrupt.
- In the transmit interrupt, check if there is data in your FIFO. If so, move it to the UART FIFO and keep the interrupt enabled. If there is nothing to put in the FIFO, disable the transmit interrupt.

Pins Options

The RP2040 has a somewhat flexible mapping of pins for the serial interfaces (UART, SPI and I2C).

The options for UART0 are:

Function	GPIOs
Tx	0, 12, 16, 28
Rx	1, 13, 17, 29
CTS	2, 14, 18
RTS	3, 15, 19

The options for UART1 are:

Function	GPIOs
Tx	4, 8, 20, 24
Rx	5, 9, 21, 25
CTS	6, 10, 22, 26
RTS	7, 11, 23, 27

Selected SDK Functions

These functions are the library `hardware_uart`.

In this functions, `uart` should be `uart0` or `uart 1`.

```
uint uart_init (uart_inst_t *uart, uint baudrate)
```

Initialize a UART, `baudrate` is in bps. Must be called before the other functions. Returns the actual baudrate programmed.

```
uint uart_set_baudrate (uart_inst_t *uart, uint baudrate)
```

Change the baudrate of a UART. Returns the actual baudrate programmed.

```
static void uart_set_hw_flow (uart_inst_t *uart, bool cts, bool rts)
```

Turns on or off the hardware flow control options.

```
static void uart_set_format (uart_inst_t *uart, uint data_bits, uint stop_bits, uart_-
parity_t parity)
```

Set the format of the data sent and received:

- `data_bits` must be between 5 and 8
- `stop_bits` must be 1 or 2
- `parity` must be one of the following: `UART_PARITY_NONE`, `UART_PARITY_EVEN`, `UART_PARITY_ODD`

```
static void uart_set_irq_enables (uart_inst_t *uart, bool rx_has_data, bool tx_needs_data)
```

Controls the use of the UART interrupts. If `rx_has_data` is true, enables the receive interrupt (there is data in the RX FIFO). If `tx_needs_data` is true, enables the transmit interrupt (the TX FIFO needs data).

Notice that there is no control (in the SDK) over the thresholds of the FIFO. Enabling the receive interrupt will also enable the receive timeout interrupt.

```
static void uart_set_fifo_enabled (uart_inst_t *uart, bool enabled)
```

Enables or disables the FIFOs in the UART. The RP2040 does not allow independent control over RX and TX FIFOs, you can have both or none.

```
static bool uart_is_readable (uart_inst_t *uart)
```

Return true if there is data in the receive FIFO.

```
bool uart_is_readable_within_us (uart_inst_t *uart, uint32_t us)
```

Wait at most `us` microseconds for data to be available in the receive FIFO. Return true if data is available or false if the time expired with no data available.

```
static bool uart_is_writable (uart_inst_t *uart)
```

Return true if there is space available in the TX FIFO.

```
static void uart_tx_wait_blocking (uart_inst_t *uart)
```

Blocks until the TX FIFO and the transmit shift register are empty.

```
static void uart_putc_raw (uart_inst_t *uart, char c)
```

Waits for space in the TX FIFO and puts a character in it.

Notes:

- Does not perform CR/LF conversion.
- The function return when the character is put in the FIFO, not when it is sent (this can take same time if there are more characters in the FIFO and/or hardware flow control is used).

```
static void uart_putc (uart_inst_t *uart, char c)
```

Waits for space in the TX FIFO and puts a character in it.

Notes:

- If CR/LF conversion is active (see `uart_set_translate_crlf`) and `c` is a line feed (0x0A), this function will put two characters in the TX FIFO, 0x0D (carriage return) and 0x0A. It will wait for space in the FIFO before putting each one.

- The function return when the character is put in the FIFO, not when it is sent (this can take same time if there are more characters in the FIFO and/or hardware flow control is used).

```
static void uart_puts (uart_inst_t *uart, const char *s)
```

Sends a null terminate string. The logic for each character in the string is similar to the one in `uart_putc`, except that if CR/LF conversion is active it will not insert a carriage return if the line feed is already preceded by a carriage return in the string. The ending null is not sent.

The function returns when the last character is put in the FIFO.

```
static void uart_write_blocking (uart_inst_t *uart, const uint8_t *src, size_t len)
```

Sends `len` characters starting from `src`. Does not perform CR/LF conversion. The function returns when the last character is put in the FIFO, blocking for space as necessary.

```
static char uart_getc (uart_inst_t *uart)
```

Read a character from the UART, will block until one is available in the RX FIFO.

```
static void uart_read_blocking (uart_inst_t *uart, uint8_t *dst, size_t len)
```

Reads `len` characters into `dst`, blocking as necessary for the characters to be received.

```
static void uart_set_break (uart_inst_t *uart, bool en)
```

Turns on or off the transmission of a break condition.

```
void uart_set_translate_crlf (uart_inst_t *uart, bool translate)
```

If `translate` is true, a line feed (0x0A) will be translate to carriage return (0x0D) + line feed in `uart_putc` and `uart_puts`.

```
static uint uart_get_dreq (uart_inst_t *uart, bool is_tx)
```

Return the DREQ (DMA Request) for transmitting (`is_tx = true`) or receiving (`is_tx = false`).

Using the UART Registers

The functions available in the SDK does not support all the functionality provided by the UARTs. If you want more control you will have to access the UART Registers.

The complete documentation of the registers available is in the RP2040 datasheet. Here I will just give a general idea of how this is done.

All registers are mapped into memory. The UART0 and UART1 registers start at base addresses of 0x40034000 and 0x40038000 respectively (defined as `UART0_BASE` and `UART1_BASE` in the SDK). Each register is at an offset of these base addresses.

The SDK defines routines, structures and constants that simplify accessing the UART registers, as exemplified bellow:

```

1  // Tests if Overrun Error is set in the Receive Status Register
2  bool uart_overrun_error(uart_inst_t *uart) {
3      // uart_get_hw(uart) returns the base address of the uart
4      // uart_get_hw(uart)->rsr returns the contents of the receive status register
5      return !(uart_get_hw(uart)->rsr & UART_UARTRSR_OE_BITS);
6  }
7
8  // Clear errors in the Receive Status Register
9  void uart_clear_errors(uart_inst_t *uart) {
10     // uart_get_hw(uart) returns the base address
11     // uart_get_hw(uart)->rsr access the receive status register
12     uart_get_hw(uart)->rsr = 0;    // doesn't matter what is written
13 }
14
15 // Set stick one parity
16 // (send and check parity bit as 1)
17 void uart_clear_overrun(uart_inst_t *uart) {
18     // uart_get_hw(uart) returns the base address
19     // &uart_get_hw(uart)->rsr returns the address of the receive status register
20     // hw_write_masked(&uart_get_hw(uart)->lcr_h, data, mask) write data to
21     // the bits selected by mask
22     hw_write_masked(&uart_get_hw(uart)->lcr_h,
23                     UART_UARTLCR_H_PEN_BITS |
24                     UART_UARTLCR_H_EPS_BITS |
25                     UART_UARTLCR_H_SPS_BITS,
26                     UART_UARTLCR_H_PEN_BITS |
27                     UART_UARTLCR_H_EPS_BITS |
28                     UART_UARTLCR_H_SPS_BITS);
29 }

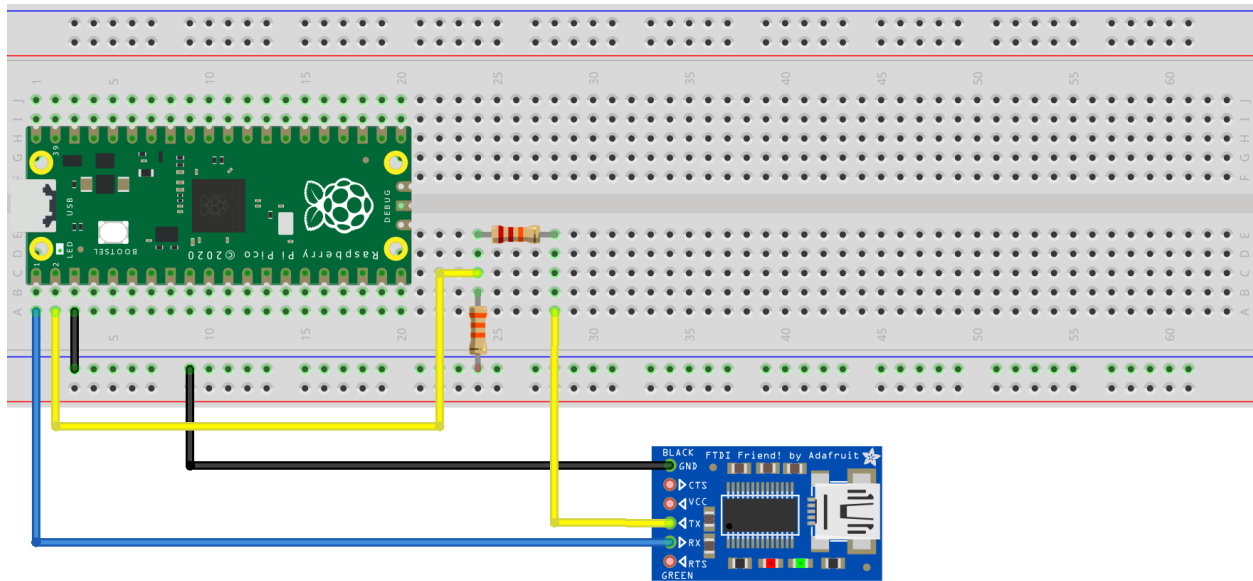
```

The definition of the structures and constants can be found at `pico-sdk\src\rp2040`.

Example

To use this example you need to connect a Pi Pico to a PC.

As modern PCs do not have serial interfaces, you will need a “TTL” serial to USB adapter (sometimes called an “FTDI adapter”, FTDI is a company that makes serial to USB chips). Care must be taken in that the the RP2040 works at 3.3V and will be damaged if 5V (the standard voltage for TTL chips) is applied at any pin. Very few adapters have the option to use 3.3V in the Rx and Tx pin. You can use a resistive divisor to reduce to 3.3V the voltage inputted in the Rx pin in the Pi Pico:



fritzing

Connecting a Serial to USB Adapter

You can also use a Pi Pico as a simple serial to USB converter, see the examples in the USB chapter.

On the software side in the PC, depending on the adapter and OS you may need a driver; this should be provided by the vendor of the adapter. You will also need a communication program that sends the characters you type and shows on the screen the received characters. A simple option is the Serial Monitor in the Arduino IDE. For Windows, PuTTY (available at putty.org) is a popular option. For Linux, you may use minicom.

In this example the communication parameters are 300bps (very slow, so you can see the characters arriving), 8 data bits, no parity and 1 stop bit ("8N1").

The program will sum decimal numbers:

- The receive interrupt will be used to input decimal numbers, a carriage return (Enter) will signal the end of the number input. The digits will be echoed (sent back) in the interrupt. When Enter is received, the interrupt is disabled, so received characters will be stored in the FIFO while the sum is updated.
- In the main program we will wait for the input of a number, update the sum, transmit it and re-enable the receive interrupt.

UART Example

```
1  /**
2   * @file uartsum.c
3   * @author Daniel Quadros
4   * @brief Example of using the UART
5   * @version 0.1
6   * @date 2022-06-17
7   *
8   * @copyright Copyright (c) 2022, Daniel Quadros
9   *
10  */
11
12  #include "stdio.h"
13  #include "pico/stdlib.h"
14  #include "hardware/uart.h"
15  #include "hardware/irq.h"
16
17  // Select UART and Pins
18  #define UART_ID uart0
19  #define UART_TX_PIN 0
20  #define UART_RX_PIN 1
21
22  // UART Configuration
23  #define BAUD_RATE 300
24  #define DATA_BITS 8
25  #define STOP_BITS 1
26  #define PARITY UART_PARITY_NONE
27
28  // UART interrupt request
29  int UART_IRQ;
30
31  // Current number and sum
32  volatile int number;
33  volatile bool number_received = false;
34  volatile int sum = 0;
35
36  // Rx interrupt handler
37  void on_uart_rx() {
38      // There can be multiple chars in the FIFO
39      while (uart_is_readable(UART_ID)) {
40          uint8_t ch = uart_getc(UART_ID);
41
42          if (ch == 0x0D) {
```

```
43         // A number was entered
44         // disable interrupt and signal number received
45         irq_set_enabled(UART_IRQ, false);
46         number_received = true;
47         break;
48     } else if ((ch >= '0') && (ch <= '9')) {
49         // Update number, limit to 4 digits
50         number = (number*10 + ch - '0') % 10000;
51         // Echo the digit
52         if (uart_is_writable(UART_ID)) {
53             uart_putc(UART_ID, ch);
54         }
55     }
56 }
57 }
58
59 // Main Program
60 int main() {
61     char msg[30]; // Buffer for sum message
62
63     // Set up UART
64     uart_init(UART_ID, BAUD_RATE);
65     uart_set_hw_flow(UART_ID, false, false);
66     uart_set_format(UART_ID, DATA_BITS, STOP_BITS, PARITY);
67     uart_set_fifo_enabled(UART_ID, true);
68
69     // Set the TX and RX pins
70     gpio_set_function(UART_TX_PIN, GPIO_FUNC_UART);
71     gpio_set_function(UART_RX_PIN, GPIO_FUNC_UART);
72
73     // Set up and enable receive interrupt
74     UART_IRQ = UART_ID == uart0 ? UART0_IRQ : UART1_IRQ;
75     irq_set_exclusive_handler(UART_IRQ, on_uart_rx);
76     irq_set_enabled(UART_IRQ, true);
77     uart_set_irq_enables(UART_ID, true, false);
78
79     // Main loop
80     while (1) {
81         if (number_received) {
82             // update sum
83             sum = (sum + number) % 1000000; // limit to 6 digits
84
85             // set up the sum message
```

```
86         sprintf (msg, " Sum:%d\r\n", sum);
87
88         // send sum
89         uart_puts(UART_ID, msg);
90
91         // wait for space in the Tx FIFO, so we can echo received chars
92         while (!uart_is_writable(UART_ID)) {
93         }
94
95         // get ready to receive another number
96         number = 0;
97         number_received = false;
98         irq_set_enabled(UART_IRQ, true);
99     }
100 }
101
102 }
```

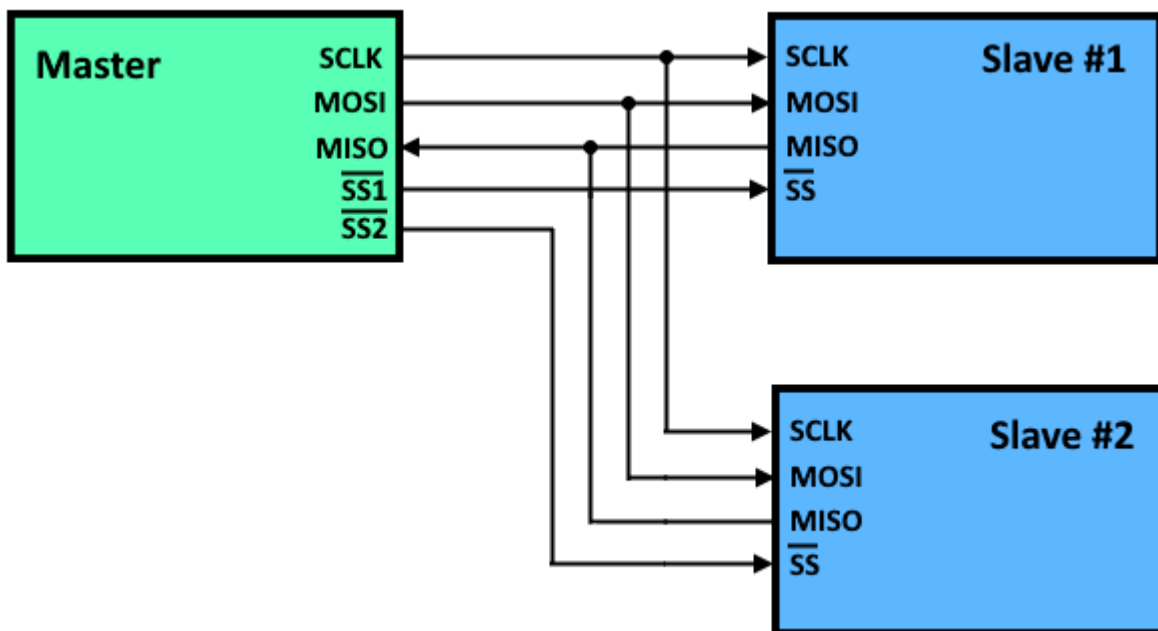
Communication Using SPI

SPI (*Serial Peripheral Interface*) is a very popular electrical protocol for connecting all kind of devices to microcontrollers, particularly when high speed is needed (like SD cards and LCD displays).

SPI Basics

SPI is notable for simultaneously transferring data in both directions with a single clock: one bit is sent and one bit is received with each clock pulse. In situations where you only want to receive some data you still send something (zeros is a common value but some devices require specific values).

It uses a master/slave multi-drop topology where the *master* generates the clock and asserts a signal that selects the *slave*. Multiple slaves can be connected to the same data and clock lines of a master, but each has a separated selection signal.



SPI topology

SPI Signals

SPI uses four signals:

- **SCLK:** is the *serial clock* (an output for the master and input for the slaves)
- **MOSI:** the *master out slave in* data signal (an output for the master and input for the slaves)
- **MISO:** the *master in slave out* data signal (an input for the master, output for the selected slave and high-impedance for non-selected slaves)
- **SS:** the *slave select* data signal (an output for the master and input for the slaves). Each slave has a separated SS signal. In most cases this is an *active low* signal: it is normally HIGH, a LOW level asserts the selection.

Some devices use other names for this signals, like SCK, DI, DO and CS.

You will also see references to *3-wire SPI*. This is a half-duplex electrical protocol where MOSI and MISO is combined in a single bi-directional signal. The RP2040 SPI peripheral does not support this use (but it can be easily implemented with the PIO).

SPI Modes

SPI has four *modes* based in what is the idle state of the SCLK line and what edge of SCLK is used to clock the data out and in. This characteristics are called *clock polarity* (CPOL) and *clock phase* (CPHA):

- CPOL=0 means that SCLK idles at LOW level.
- CPOL=1 means that SCLK idles at HIGH level.
- CPHA=0 means that the “out” side changes the data on the trailing edge of the preceding clock cycle (or before the first cycle if it is the first bit), while the “in” side captures the data on the leading edge of the clock cycle.
- CPHA=1 means that the “out” side changes the data on the leading edge of the clock cycle, while the “in” side captures the data on the trailing edge of the clock cycle.

It is common to refer to this combinations by a *mode number*:

Mode	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1

SPI in the RP2040

The RP2040 has two SPI peripherals with the following features:

- Can be used as master or slave

- Support data size of 4 to 16 bits
- Has 8 position FIFOs for reception and transmission
- Support the four SPI modes
- Flexible clock generation
- Can generate interrupts and work with DMA

The SPI peripheral has also support for two less common SPI variants (Texas Instruments synchronous serial interface and National Semiconductor Microwire) that I won't describe here.

The RP2040 datasheet and C/C++ SDK use the following names for the SPI signals:

- SCLK: SSPCLK / SCK
- MISO: SSPRXD / RX
- MOSI: SSPTXD / TX
- SS: SSPFSS / CSN

Clock Generation

The clock for master mode is derived from `clk_peri` by a two stage divisor. The first stage divides `clk_peri` by a factor of 2 to 254 (in steps of two). The second stage divides the resulting frequency by a factor of 1 to 256.

Pins Options

The RP2040 has a somewhat flexible mapping of pins for the serial interfaces (UART, SPI and I2C).

The options for SPI0 are:

Function	GPIOs
SCLK	2, 6, 18, 22
MISO	0, 4, 16, 20
MOSI	3, 7, 19, 23
SS	1, 5, 17, 21

The options for SPI1 are:

Function	GPIOs
SCLK	10, 14, 26
MISO	8, 12, 24, 28
MOSI	11, 15, 27
SS	9, 13, 25, 29

Note that the SS pin is only relevant when operating in slave mode (the peripheral will automatically test it). In master mode you have to manually control the slave selection, it can be any digital output

pin. When you are using SPI to communicate with a single slave it is customary (but no obligatory) to use one of the SS pin of the peripheral.

Selected SDK Functions

These functions are in `hardware_spilibrary`. The `spi` parameter should be `spi0` or `spi1`.

```
uint spi_init (spi_inst_t *spi, uint baudrate)
```

Initializes a SPI interface. This function must be called before the others.

The interface is put in master mode and the clock is set to the closer value to baudrate available. If you want to operate as slave, call `spi_set_slave()` after this function.

Returns the actual baudrate.

```
uint spi_set_baudrate (spi_inst_t *spi, uint baudrate)
```

Sets clock to the closer value to baudrate available.

Returns the actual baudrate.

```
static void spi_set_format (spi_inst_t *spi, uint data_bits, spi_cpol_t cpol, spi_cpha_t cpha, __unused spi_order_t order)
```

Configures the format for a SPI interface:

- `data_bits`: 4 to 16
- `cpol`: clock polarity (`SPI_CPOL_0` or `SPI_CPOL_1`)
- `cpha`: clock phase (`SPI_CPHA_0` or `SPI_CPHA_1`)
- `order`: must be `SPI_MSB_FIRST`

```
static void spi_set_slave (spi_inst_t *spi, bool slave)
```

Selects between master (`slave false`) and slave (`slave true`) mode.

```
static bool spi_is_writable (const spi_inst_t *spi)
```

Returns true if there is space in the transmission FIFO.

```
static bool spi_is_readable (const spi_inst_t *spi)
```

Returns true if there is data in the reception FIFO.

```
static bool spi_is_busy (const spi_inst_t *spi)
```

Returns true if the SPI is transmitting and/or receiving a frame or the transmit FIFO is not empty. False means that no data is transferring and no data is waiting in the FIFO to be transmitted.

```
int spi_write_read_blocking (spi_inst_t *spi, const uint8_t *src, uint8_t *dst, size_t len)
```



```
int spi_write16_read16_blocking (spi_inst_t *spi, const uint16_t *src, uint16_t *dst,
size_t len)
```

Write `len` items from `src` and, simultaneously, read `len` items to `dst`. Blocks until all data is transferred.

The first version uses byte buffers and is for data length up to 8 bits. In the second version the buffers hold 16 bit values.

Returns the number of items transfered.

```
int spi_write_blocking (spi_inst_t *spi, const uint8_t *src, size_t len)
int spi_write16_blocking (spi_inst_t *spi, const uint16_t *src, size_t len)
```

Write `len` items from `src` and ignore the received data. Blocks until all data is transferred.

The first version uses a byte buffer and is for data length up to 8 bits. In the second version the buffer hold 16 bit values.

Returns the number of items transferred.

```
int spi_read_blocking (spi_inst_t *spi, uint8_t repeated_tx_data, uint8_t *dst, size_t
len)
int spi_read16_blocking (spi_inst_t *spi, uint16_t repeated_tx_data, uint16_t *dst,
size_t len)
```

Read `len` items into `dst`, sending `len` items equal to `repeated_tx_data`. Blocks until all data is transferred.

The first version uses a byte buffer and is for data length up to 8 bits. In the second version the buffer hold 16 bit values.

Returns the number of items transferred.

```
static uint spi_get_dreq (spi_inst_t *spi, bool is_tx)
```

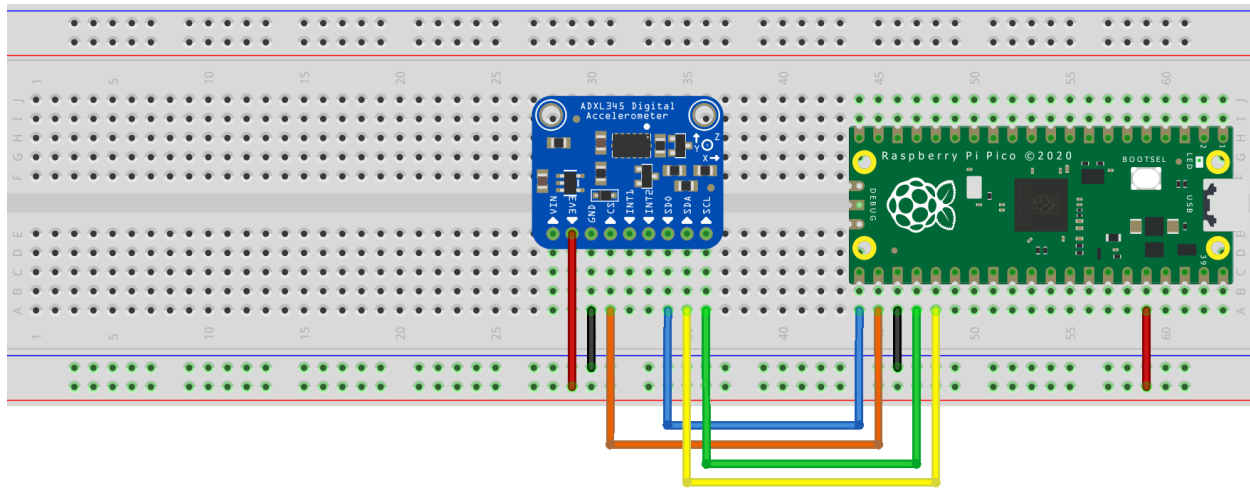
Returns the DREQ to use for transferring data via DMA to/from the SPI peripheral. `is_tx` specifies the direction (true = transfer data to transmit, false = transfer received data).

Example

In this example the RP2040 will be a SPI master communicating with an ADXL345 accelerometer. Note that the objective here is to show the SPI communication, not how to use the ASXL345 (detailed information on it can be found in its datasheet).

Note: The ADXL345 can work in I²C, SPI and “3-wire SPI”, depending on its chip select signal and programming. When working in I²C mode the SO pin selects the address. Because of this, some ADXL345 boards will have SO connected to ground or Vcc, this connection *must* be broken for SPI operation.

The following figure shows the connections (your ADXL345 board may have a different layout, check the documentation and be alert to SO tied to ground or Vcc).



fritzing

ADXL345 connections to the Raspberry Pi Pico

SPI Example

```

1  /**
2   * @file adxl345.c
3   * @author Daniel Quadros
4   * @brief Example of using the SPI to interface an ADXL345 accelerometer
5   *        Details on the ADXL345 can be found in its datasheet
6   * @version 0.1
7   * @date 2022-07-27
8   *
9   * @copyright Copyright (c) 2022, Daniel Quadros
10  *
11  */
12
13  #include "stdio.h"
14  #include "pico/stdlib.h"
15  #include "hardware/spi.h"
16
17  // Select SPI and Pins
18  #define SPI_ID spi0
19  #define SPI_SCLK_PIN 18
20  #define SPI_MISO_PIN 16
21  #define SPI_MOSI_PIN 19
22  #define SPI_SS_PIN 17
23

```

```

24 // SPI Configuration
25 #define BAUD_RATE 1000000 // 1 MHz
26 #define DATA_BITS 8
27
28 // ADXL345 Registers
29 #define DEVID 0x00
30 #define BW_RATE 0x2C
31 #define POWER_CTL 0x2D
32 #define DATA_FORMAT 0x31
33 #define DATA_X0 0x32
34
35 // This bits are ORed to the register address
36 #define READ_BIT 0x80 // this is a read
37 #define MULTI_BIT 0x40 // multiple bytes are transfered
38
39 // Structure to hold raw acceleration values
40 typedef struct
41 {
42     int x;
43     int y;
44     int z;
45 } AccelRaw;
46
47
48 // Local routines
49 static void ADXL345_init (void);
50 static uint8_t ADXL345_readId(void);
51 static void ADXL345_readAccel(AccelRaw *raw);
52
53 // Assert the SS signal
54 static inline void ss_select() {
55     asm volatile("nop \n nop \n nop");
56     gpio_put(SPI_SS_PIN, 0); // Active low
57     asm volatile("nop \n nop \n nop");
58 }
59
60 // Remove SS signal
61 static inline void ss_deselect() {
62     asm volatile("nop \n nop \n nop");
63     gpio_put(SPI_SS_PIN, 1);
64     asm volatile("nop \n nop \n nop");
65 }
66

```

```

67 // Main Program
68 int main() {
69     // Start stdio and wait for USB connection
70     stdio_init_all();
71     while (!stdio_usb_connected()) {
72         sleep_ms(100);
73     }
74     printf("Hello, ADXL345!\n");
75
76     // Set up the SS pin
77     gpio_init(SPI_SS_PIN);
78     gpio_set_dir(SPI_SS_PIN, GPIO_OUT);
79     gpio_put(SPI_SS_PIN, 1);
80
81     // Set up SPI
82     uint baud = spi_init (SPI_ID, BAUD_RATE);
83     printf ("SPI @ %u Hz\n", baud);
84     spi_set_format (SPI_ID, DATA_BITS, SPI_CPOL_1, SPI_CPHA_1, SPI_MSB_FIRST);
85
86     // Set up the SPI pins
87     gpio_set_function(SPI_SCLK_PIN, GPIO_FUNC_SPI);
88     gpio_set_function(SPI_MISO_PIN, GPIO_FUNC_SPI);
89     gpio_set_function(SPI_MOSI_PIN, GPIO_FUNC_SPI);
90
91     // Init the ADXL345
92     ADXL345_init ();
93
94     // Report ADXL345 identification
95     printf ("ID = %o\n", ADXL345_readId());
96
97     // Main loop
98     AccelRaw raw;
99     while (1) {
100         ADXL345_readAccel(&raw);
101         printf ("Accel X=%d Y=%d Z=%d\n", raw.x, raw.y, raw.z);
102         sleep_ms(1000);
103     }
104
105 }
106
107 // Initialize ADXL345
108 static void ADXL345_init () {
109     uint8_t buf[2];

```

```

110
111     // Turn off LOW_POWER and select sample rate
112     buf[0] = BW_RATE;
113     buf[1] = 0x0F;      // Maximum sample rate
114     ss_select();
115     spi_write_blocking(SPI_ID, buf, 2);
116     ss_deselect();
117
118     // Select data format
119     buf[0] = DATA_FORMAT;
120     buf[1] = 0x0B;  //4wire SPI +/- 16g range, 13-bit resolution
121     ss_select();
122     spi_write_blocking(SPI_ID, buf, 2);
123     ss_deselect();
124
125     // Start measurements
126     buf[0] = POWER_CTL;
127     buf[1] = 0x08;
128     ss_select();
129     spi_write_blocking(SPI_ID, buf, 2);
130     ss_deselect();
131 }
132
133 // Reads ADXL345 identification
134 static uint8_t ADXL345_readId() {
135     uint8_t bufTx[] = { DEVID | READ_BIT, 0x00 };
136     uint8_t bufRx[2] = { 0x55, 0x55 };
137
138     ss_select();
139     spi_write_read_blocking (SPI_ID, bufTx, bufRx, 2);
140     ss_deselect();
141
142     return bufRx[1];
143 }
144
145 // Reads raw acceleration data
146 static void ADXL345_readAccel(AccelRaw *raw) {
147     uint8_t selReg[] = { DATA0 | READ_BIT | MULTI_BIT};
148     uint8_t buf[6];
149
150
151     ss_select();
152     spi_write_blocking (SPI_ID, selReg, 1);    // Selects first register

```

```
153     spi_read_blocking (SPI_ID, 0x00, buf, 6);    // Reads 6 registers
154     ss_deselect();
155
156     raw->x = (((int)buf[1]) << 8) | buf[0];
157     raw->y = (((int)buf[3]) << 8) | buf[2];
158     raw->z = (((int)buf[5]) << 8) | buf[4];
159 }
```

There are a few points I would like to highlight in the code:

- Notice that the SS signal is managed by the gpio functions. Short sequences of NOPs are used to create a short delay before and after changing this signal.
- The SPI initialization requires calls to `spi_init()`, `spi_set_format` and to `gpio_set_function` (for each pin). The ADXL345 uses SPI mode 3.
- The ID should be octal 345.
- The ADXL345 is organized in registers. At the start of each communication it expects a register address, plus two bits that inform if we are reading or writing to the register and if this is a multi-byte operation. In multi-byte operations the register address is incremented with each byte transferred.
- To configure the ADXL345 in `ADXL345_init()` we use `spi_write_blocking` as the ADXL345 will not replay anything.
- To read the ID two bytes needs to be transferred. The first one is the address of the ID register (plus the READ bit); it is sent from the RP2040 to the ADXL345. The second byte is the ID, sent from the ADXL345 to the RP2040. I used a single `spi_write_read_blocking()` call to do both, I could also use a `spi_write_blocking()` followed by a `spi_read_blocking()`, as long as I didn't change the SS line between the two calls.
- To read the raw acceleration values (`ADXL345_readAccel()`) we first write the address of the first result register plus the bits that indicate that this is a read and we gonna read multiple bytes. Then we read the six bytes using `spi_read_blocking()`, as the value sent to the ADX345 is irrelevant.

Analog Input: the ADC

Overview

The ADC (Analog to Digital Converter) is used to measure an analog voltage. The result of the reading is a binary number that is proportional to the voltage.

The ADC in the RP2040 is a SAR ADC (Successive Approximation Register), where the bits of the 12 bit result are generated one at a time (this is transparent to the programmer). It uses a 48 MHz clocks and can do up to 500 thousand samples per second.

The maximum result plus one (4096) corresponds to an external reference voltage. In the Raspberry Pi Pico, this reference voltage is the same 3.3V that powers the chip. Damage can occur if a voltage greater than the reference is applied to an analog input.

There is only one ADC in the RP2040, but it has five inputs (or *channels*). One is an internal temperature sensor, the other four are connected to the same pins as GPIO26 to GPIO29. In the Pi Pico, only three of these pins are available in the castellated pins. You can selected one input at a time or enable a mode where the channels are automatic changed after each reading.

When a reading is completed, the result can be put in a four element FIFO (First In First Out queue) where it can be read by the ARM processors. An interrupt can be generated when the FIFO reaches a configurable level.

Modes of Operation

There are two modes of operation: *one-shot* and *free-running*.

In the *one-shot* mode, the program starts each reading. The ADC input should be selected before the conversion starts.

In the *free-running* mode, conversions are started automatically at regular intervals. By default, a new conversion starts as soon as the previous one is finished (96 cycles, 2 microseconds with the 48 MHz clock). A divisor (with 16 bits for the integer part and 8 bits for the fractional part) can be applied to reduce the conversion rate. If the fractional part is zero, each conversions will take (divisor+1) cycles. If the fractional part is not zero, conversions will take (int(divisor)+1) or (int(divisor)+2) cycles in such a way that the average will be (divisor+1) cycles.

For example, a divisor of 191 will result in 192 cycles per conversion (the double of the default). A divisor of 191.5 will alternate equally between 192 and 193 cycles, resulting in a average of 192.5. If a divisor of 191.25 is used, we will have one 193 cycles delay for each three 192 cycles delay, so the average will be 192.25.

A zero divisor causes default behavior.

In free running mode we can automatically sample multiple inputs. A mask defines which of the 5 inputs should be used. The inputs corresponding to '1' bits will be selected in order, after the conversion for the current selection finishes (this initial selection need not to correspond to a set bit). For example, a mask of 10011 indicates that channels 0, 1 and 4 will be used (in this order).

Accuracy of the ADC

When using the ADC we need to keep in mind that there are a number of motives, some inside e some outside the RP2040, for errors in the result. Some of them are:

- Imprecision in the outside circuit used to scale the measured signal to the range of the ADC.
- The outside circuit and/or the ADC input affecting the original signal.
- Electrical noise.
- Non-linearity of the ADC.
- Imprecision and/or variations in the reference voltage.

While the result of the RP2040 ADC have 12 bits, the datasheet gives the *Effective Number Of Bits* (ENOB) as 8.7. This means that only about 9 bits are reliable (considering only errors internal to the RP2040).

Temperature Sensor

The RP2040 includes an internal temperature sensor, connected to channel 4 of the ADC. The expected voltage given by the sensor is 0.706V at 27 degrees C; this voltage will drop 1.721mV per additional degree C, which suggests the formula

$$T = 27 - (\text{voltage} - 0.706)/0.001721$$

Unfortunately, this may not work in most cases as:

- The values can change from device to device
- The drop per degree is not constant, it will change with the temperature
- As the drop per degree is low, small differences in the reference voltage will result in significant difference in the calculated temperature (the RP2040 datasheet mentions a 4 degree difference for a 1% change in the reference voltage).

As the sensor is inside the chip, it will measure the chip's temperature, not the ambient temperature. So, for most applications, it is not a substitute for an external temperature sensor.

The temperature sensor must be enabled (powered on) before use.

Selected SDK Functions

The functions for the adc are in the library `hardware_adc`.

```
void adc_init (void)
```

Initializes the ADC hardware.

```
static void adc_gpio_init (uint gpio)
```

Prepares a GPIO pin to be used as an ADC input (disables all digital functions). `gpio` must be between 26 and 29.

```
static void adc_select_input (uint input)
```

Select the ADC input channel. Channels 0 to 3 correspond to GPIO 26 to GPIO29, channel 4 is the temperature sensor.

```
static uint adc_get_selected_input (void)
```

Returns the currently select ADC input channel (0 to 4).

```
static void adc_set_round_robin (uint input_mask)
```

`input_mask` should be between 0 and 0x1F. If `input_mask` is zero, round robin will be disabled. If not zero, round robin is enabled and bit 1 in `input_mask` indicate the channels to be sampled.

```
static void adc_set_temp_sensor_enabled (bool enable)
```

If `enable` is true the temperature sensor will be powered up. If `enable` is false it will be turned off.

```
static uint16_t adc_read (void)
```

Performs an ADC conversion, using single-shot mode. Waits for the result and returns it.

```
static void adc_run (bool run)
```

Enables (`run` true) or disables (`run` false) the free-running sampling mode.

```
static void adc_set_clkdiv (float clkdiv)
```

Sets the ADC clock divisor. The time between samples will be `(1+clkdiv) cycles` (if `clkdiv` is less than 95 the function will use 95, as the minimum time is 96 cycles).

```
static void adc_fifo_setup (bool en, bool dreq_en, uint16_t dreq_thresh, bool err_in_fifo, bool byte_shift)
```

Configures the ADC FIFO:

- if `en` is true, results are placed in the FIFO.
- if `dreq_en`, DMA requests will be generated when there is results in the ADC FIFO.
- `dreq_thresh` defines how many results need to be in the ADC FIFO for a DMA or IRQ request be generated.

- if `err_in_fifo` is true, bit 15 in the results will indicate if an error occurred during conversion.
- if `byte_shift` is true, the results in ADC FIFO are shift right 4 bits (discarding the lower four bits). This can be useful if you do not need the full precision and want to DMA results to a byte buffer (each byte in the buffer will be one result).

```
static bool adc_fifo_is_empty (void)
```

Returns true if there is no result in the ADC FIFO.

```
static uint8_t adc_fifo_get_level (void)
```

Returns the number of results in the DAC FIFO.

```
static uint16_t adc_fifo_get (void)
```

Gets an ADC result from the FIFO. It is unspecified what you get if the FIFO is empty.

```
static uint16_t adc_fifo_get_blocking (void)
```

Waits (blocking) until there is data in the ADC FIFO and returns the first result.

```
static void adc_fifo_drain (void)
```

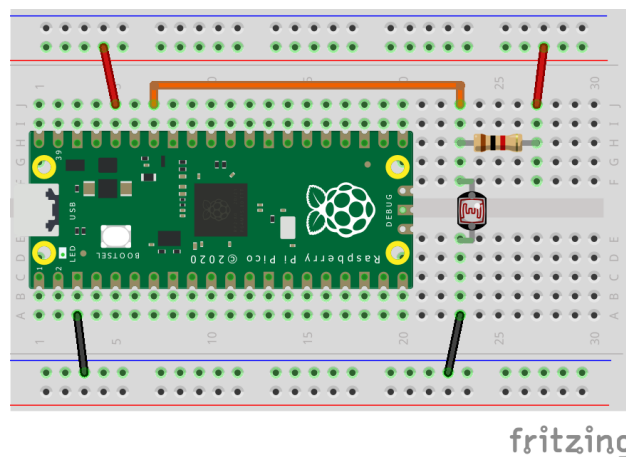
Waits for the current conversion (if any) to complete and empties the ADC FIFO (discarding any results there).

```
static void adc_irq_set_enabled (bool enabled)
```

Enables (enabled true) or disables (enabled false) the ADC interrupts.

Example

The following example uses the free-running mode and round-robin sampling to measure the internal temperature sensor and an external light sensor. The light sensor is just an LDR and a 1k resistor:



Circuit for ADC example

Here is the code:

ADC Example

```
1  /**
2   * @file adcdemo.c
3   * @author Daniel Quadros
4   * @brief Example of using the ADC in the RP2040 to read
5   *        the internal temperature sensor and a externa light sensor
6   * @version 0.1
7   * @date 2022-07-06
8   *
9   * @copyright Copyright (c) 2022, Daniel Quadros
10  *
11  */
12
13  #include <stdio.h>
14  #include <string.h>
15  #include <stdlib.h>
16
17  #include "pico/stdlib.h"
18  #include "hardware/gpio.h"
19  #include "hardware/adc.h"
20
21  // Where the LDR is connected
22  #define GPIO_LDR      28
23  #define ADC_INPUT_LDR  2
24
25  // Internal temperature sensor
26  #define ADC_INPUT_TEMPSENSOR 4
27
28  // Factor to convert ADC reading to voltage
29  // Assumes 12-bit, ADC_VREF = 3.3V
30  const float conversionFactor = 3.3f / (1 << 12);
31
32  // Main Program
33  int main() {
34      // Init stdio
35      stdio_init_all();
36      while (!stdio_usb_connected()) {
37          sleep_ms(100);
38      }
39      printf("\nADC Example\n");
40  }
```

```
41 // Init ADC
42 adc_init();
43 adc_set_temp_sensor_enabled(true);
44 adc_set_round_robin ((1 << ADC_INPUT_TEMPSENSOR) | (1 << ADC_INPUT_LDR));
45 adc_select_input (ADC_INPUT_LDR);
46
47 // Reduce the sampling to 1 ms between readings
48 float clkdiv = 0.001f * 48000000.0f - 1;
49 adc_set_clkdiv(clkdiv);
50 adc_fifo_setup (true, false, 0, false, false);
51
52 // Make sure GPIO is high-impedance, no pullups etc
53 adc_gpio_init(GPIO_LDR);
54
55 // Start the ADC
56 adc_run(true);
57
58 // Main loop
59 const int MAX_COUNT = 500;
60 float tempSum, ldrSum;
61 while (1) {
62     tempSum = 0.0f;
63     ldrSum = 0.0f;
64     for (int count = 0; count < MAX_COUNT; count++) {
65         ldrSum += adc_fifo_get_blocking() * conversionFactor;
66         tempSum += adc_fifo_get_blocking() * conversionFactor;
67     }
68     float ldrV = ldrSum/MAX_COUNT;
69     float tempC = 27.0f - (tempSum/MAX_COUNT - 0.706f) / 0.001721f;
70
71     // Print out the averages
72     printf("LDR voltage: %.2f V Temperature: %.2f\n", ldrV, tempC);
73 }
74 }
```

A Brief Introduction to the USB Controller

USB (Universal Serial Bus) is an industry standard for the connection of peripheral to hosts. While simple from a electrical viewpoint (particularly for the USB 1.1 supported by the RP2040), the protocol used is very complex.

In this chapter I will not delve deeply into the protocol itself, but take a brief look into it and how the RP2040 and its SDK supports USB in hardware and software.

Thanks to the tinyUSB library, programmers use the USB Controller without worrying too much about the USB protocol.

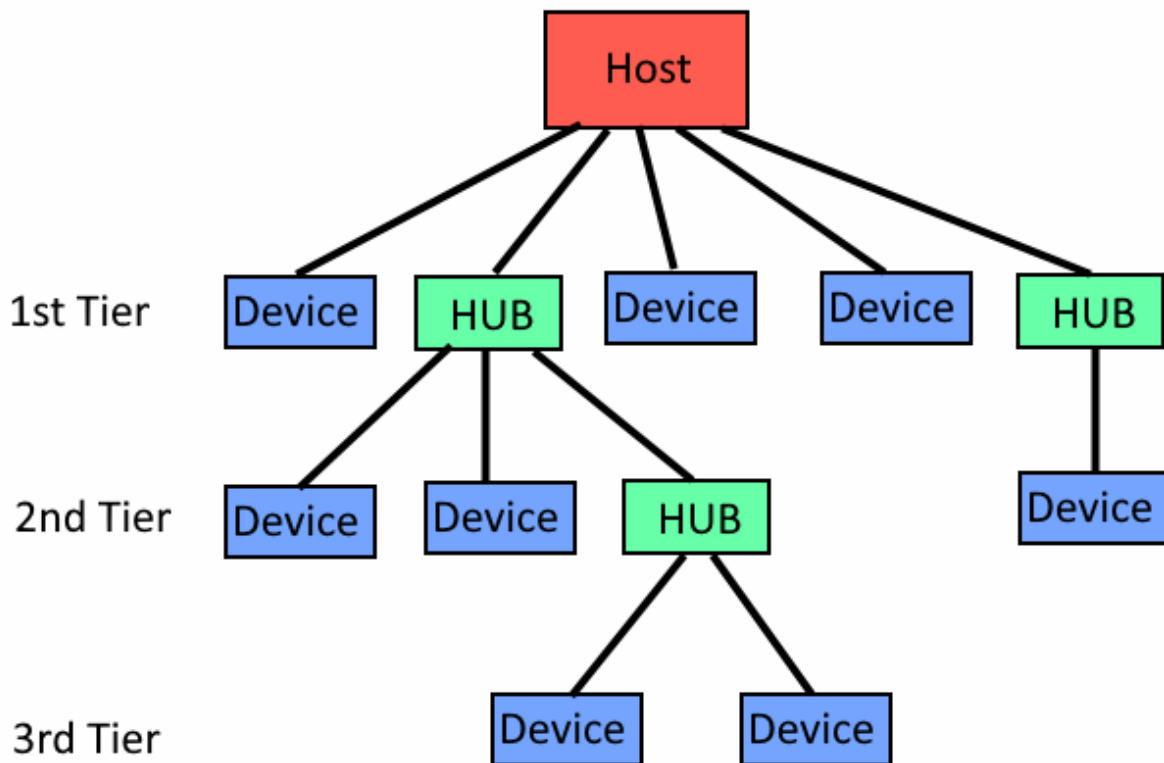
USB Basics

Unlike previous serial and parallel interfaces in PCs, USB aims to support a great range of devices with the same architecture and “infrastructure” (connectors, cables and hubs). The specification has gone through several upgrades, **the RP2040 supports the features of USB 1.x** (1.0 and 1.1) that are now documented in the USB 2.0 specification.

USB 1.x supported two speeds: **low** (1.5Mbps) and **full** (12Mbps), this is what the RP2040 supports. USB 2.0 added the **high** speed (480Mbps), USB 3.0 added **superspeed** (5Gbps), latter versions added **superspeed+** (10, 20 and 40Gbps). These numbers are the maximum rate that the signals can change, actual data throughput will be less than that.

In USB 1.x and 2.0 all communication is between a **host** and a **device**, and transmission is always initiated by the host. The RP2040 can operate as a host or as a device.

A USB system has a multiple tier star topology, where a host port connects to multiple devices; special devices (**hubs**) create new tiers. A host controller supports a maximum of 127 devices in up to five tiers.



USB Topology

As all communications will involve the host, USB (as the name says) is a **bus** and its bandwidth is shared by all devices.

A USB device is assigned a **device address** by the host during its initialization. It may be a *composite device* with multiple **device functions** (logical devices) accessible by a single address. For example, a webcam may have two device functions: a video capture device and an audio capture device. Another way to implement this is as a *compound device*, where there is an internal hub; in our example, in a webcam implemented as a compound device, the internal hub, video and audio capture would each receive an address.

Inside a device there can be up to 32 **endpoints** (16 in and 16 out). Communication is based on **pipes**, logical connections between the host and an endpoint. Endpoints are numbered from 0 to 15 at initialization, with an additional address bit indicating if it is IN or OUT. Endpoint 0 (IN and OUT) is used for device configuration. Other endpoints are grouped into **interfaces**, each interface corresponds to a device function.

There are four types of transfers in the USB protocol:

- **Control:** used for the initial configuration of the device by the host and for device-specific control.

- **Interrupt:** these transfers are initiated periodically by the host (at a rate requested by the **endpoint descriptor**) and use small packets (up to 8 bytes for low speed and 64 for full speed). A typical use is for HID devices, like keyboard and mouse.
- **Bulk:** used for error-free transfer of large amounts of data, when eventual delays are acceptable. A typical use is mass storage devices.
- **Isochronous:** used when it is important to transfer data on time, but errors can be tolerated. Typical uses are video and audio data transfer.

A transfer is divided in **transactions**, each one a group of two or three **packets**, always started by a *token packet* sent by the host and, unless it is an isochronous transfer, ended by a *handshake packet* (used by the receiver to acknowledge that the previous packets were received correct). In an **OUT transaction**, the host sends a *data packet* after the token and the handshake (if used) is sent by the device. In an **IN transaction**, the device sends a *data packet* after the token and the handshake (if used) is sent by the host. The **SETUP transaction** is like the OUT transaction, except that the data packet has always 8 bytes of setup data and there is always a handshake packet.

Except for isochronous transfers, all communication has some kind of error checking (CRC and message numbering). If the receiver detects an error, the received packet is ignored; this will cause a timeout and retry by the sender.

USB has very short timeouts. Because of this, data to be sent is normally put in a transmit buffer and then “armed” to be sent on request (by the hardware) instead of being put in the buffer only after a request is received.

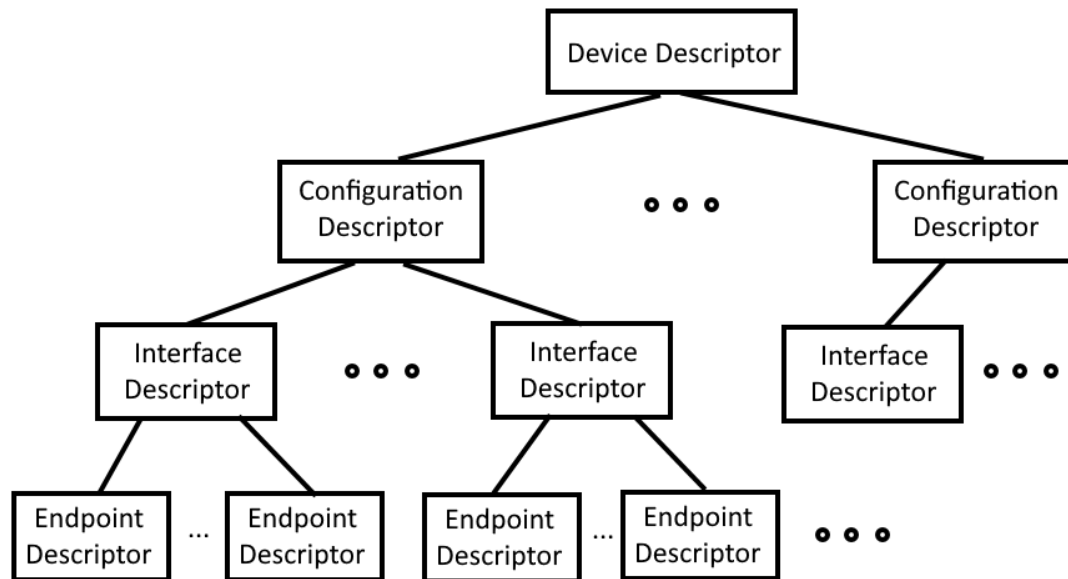
USB defines a number of **device classes**. A device class defines the messages that will be exchanged by the host and the device.

When a device is connected to the bus, a process called **enumeration** is triggered. In this process the host uses standard control transfers to:

- determine the device characteristics.
- assign an address to the device.
- get **descriptors** for the device, interfaces and endpoints. Descriptors give the host the necessary information for using its functions. There are also *string descriptors* that contain human readable descriptions; other descriptors reference text in these descriptors by an index.
- select a configuration (most devices have a single configuration).

Only after all this is done the device is ready for normal operation.

The figure below gives an idea of the relations between the various types of descriptors in a device:



Relations between USB descriptors

Among the device characteristics there is a vendor id (**VID**) and a product id (**PID**). Vendor IDs are assigned by the USB Implementers Forum (USB-IF), PIDs are assigned by the vendors. Windows (and other OSs) use VID/PID to select drivers (except for some classes like HID and MSC).

Hardware

USB 1.1 and 2.0 uses two signals for communication, named D+ and D-. The RP2040 has an integrated USB 1.1 PHY (physical driver) which interfaces the USB controller with the DP (D+) and DM (D-) pins of the chip. The PHY takes care of the electrical encoding (how bits and special conditions are represented in the D+ and D- signals).

It also contains a USB 2.0 controller that handles the low level USB protocol. It can operate in two ways:

- As a *device* (mass storage, keyboard and others) operating at full speed.
- As a *host* (like a PC) that can communicate with Low Speed and Full Speed devices.

The software must configure the USB controller, consume the received data and generate the data to be sent. A 4K RAM in the controller is used to store the configuration and data.

The two main components of the USB controller (besides the RAM) are the “Serial Rx Engine” and “Serial Tx Engine”. These engines decode and encode packets, including checking and generating CRCs (used to detected transmission errors).

Device Classes

Like mentioned above, a device class is related to the functionality of the device. It defines the messages exchanged with the host and, in some cases, the drivers that will be loaded by the operating system.

The USB-IF has defined many device classes, like the following:

- HID (Human Interface Devices): for devices such as keyboard, mouse, joystick. This class uses only control and interrupt transfers. The data exchanged must be in structures named **reports**. Some custom (simple) devices can be implemented using the HID class if all they need is to exchange small packets of data; most OSs have a API to send and receive these packets.
- CDC (Communication Device Class): for communication devices like modem and network interface. A common use is replacing serial RS232 interfaces.
- MSC (Mass Storage Class): for devices that can store large amounts of data (like pen-drives, SD cards, disk/CD/DVD drives and tape drives).
- MIDI (Musical Instrument Device Interface): for devices that use USB as the hardware transport for the MIDI protocol.

TinyUSB

TinyUSB is an open-source cross-platform USB Host/Device stack for embedded systems and is the official USB stack for the RP2040.

It implements many USB device classes, including HID, CDC, MSC and MIDI. It also allows the operation as a USB host, supporting HID, CDC and MSC devices.

TinyUSB takes care of most of the low level USB stuff and uses callbacks (routines in your code, called by tinyUSB) to inform of events that need your processing. Since RP2040 C/C++ SDK applications do not use an Operating System, your code must call periodically some routines from the tinyUSB.

The official repository is at <https://github.com/hathach/tinyusb> and is referenced by the RP2040 SDK.

The official documentation is at <https://docs.tinyusb.org/>.

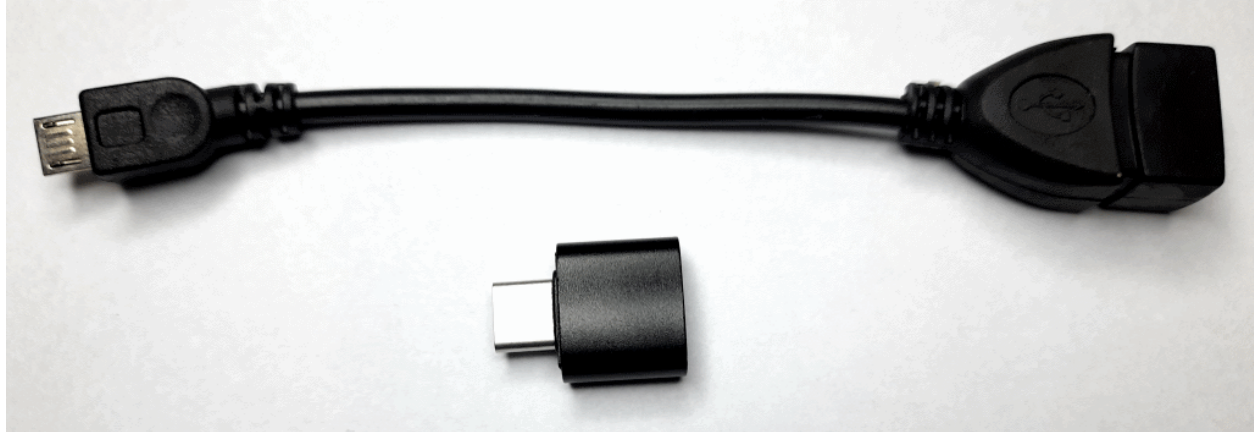
Using the USB

In the Raspberry Pi Pico (and most boards based on the RP2040) the USB pins are connected to a standard Micro-A or USB-C USB connector.

The firmware in the RP2040 ROM, activated by resetting with the BOOT signal connected to ground, implements a mass storage that is used to load an application in the Flash.

It is up to the application to implement the desired functions for the USB. A common use (supported by the SDK) is to implement a CDC device so data written by the SDK print function can be received at the PC (with appropriate drivers) as if from a serial port.

To connect a USB device to the RP2040 it must act as an USB host. A USB OTG (On The Go) adapter is necessary for the Micro-A or USB-C connector.



USB OTG Adaptors

When developing a USB device that will be a product (that is not for your personal use) there are some formalities you must address:

- You will need a **Vendor Id**. You get that from USB-IF, by paying a one-time fee (US\$6000) or becoming a USB-IF member (US\$5000/year).
- If you want to use the USB logo, you must either become a USB-IF member (US\$5000/year) or pay a license fee (US\$3500 for the first two years).

(fees values from <https://www.usb.org/getting-vendor-id> on May 7, 2022)

If these values seem too high for you, you may want to take a look at <https://github.com/obdev/v-usb/blob/master/usbdrv/USB-ID-FAQ.txt>, but take notice that USB-IF does not agree with this uses.

Most operating systems come with drivers for some device classes (like HID and MSC). If your device is not in these classes, or you want/need an specific driver, you will have to develop it (not an easy task and out of the scope of this book).

The HID Device Class

In this section I describe briefly the HID device class (with emphasis on keyboard devices), so you can follow my examples.

The HID device class tries to encompass many kinds of devices that interact with people. Besides the common keyboards, mice, joysticks and gamepads, it can support input devices that measure

some physical dimension (like length, angle, weight and temperature) and simple output devices (like alphanumeric displays).

The data is exchanged in structures called **reports**, by means of interrupt transfers (the host will periodically ask the device for reports). The host can use a `set_rate` request to ask the device to only answer requests for reports if there is a change in the report or a minimum time (the *idle rate*) has elapsed. An idle rate of zero means that the device will only send a report if there is change in the data.

To achieve generality, the format of the reports are described by *report descriptors*. These descriptors specify the length and type of the data; they can also specify the scaling of physical measurements.

Regarding keyboards, the input reports will signal the pressed keys by means of a list of keycodes or a bitmap (remember that in and out are from the host point of view).

As analyzing the reports based on the report descriptor can be complicated and a computer's keyboard must operate before a USB aware OS is loaded, the USB HID specification includes a *boot protocol* that uses a fixed and simplified report oriented to the standard PC keyboard:

- The report has 8 bytes
- The first byte is a bitmap of the state of the *modifier keys* (right and left shift, control, alt and GUI/Windows keys)
- The second byte is reserved (zero)
- The remaining 6 bytes have the keycodes of the pressed keys (zeros are used as fillers)

A consequence of this format is that the keyboard can report at most six non-modifier keys pressed ("6-key rollover"). Also the "auto-repeat" (repeating the key if its kept pressed) must be implemented in the host.

The association of keycodes to key symbols/functions can vary depending on the keyboard language and layout.

The HID host code in the tinyUSB stack will select the boot protocol and a zero idle rate when a HID device is mounted.

An output report is used to control the keyboard LEDs. Again, this task must be implemented by the host.

Example - Emulating a PC Keyboard

In this simple example we will implement a five key keyboard, supporting only the boot protocol. The first four keys will generate the codes for 1, 2, 3 and 4 keys at the top of the keyboard, the fifth key will generate the code for CAPS LOCK. The LED in the Pico board will be used as a CAPS LOCK LED.

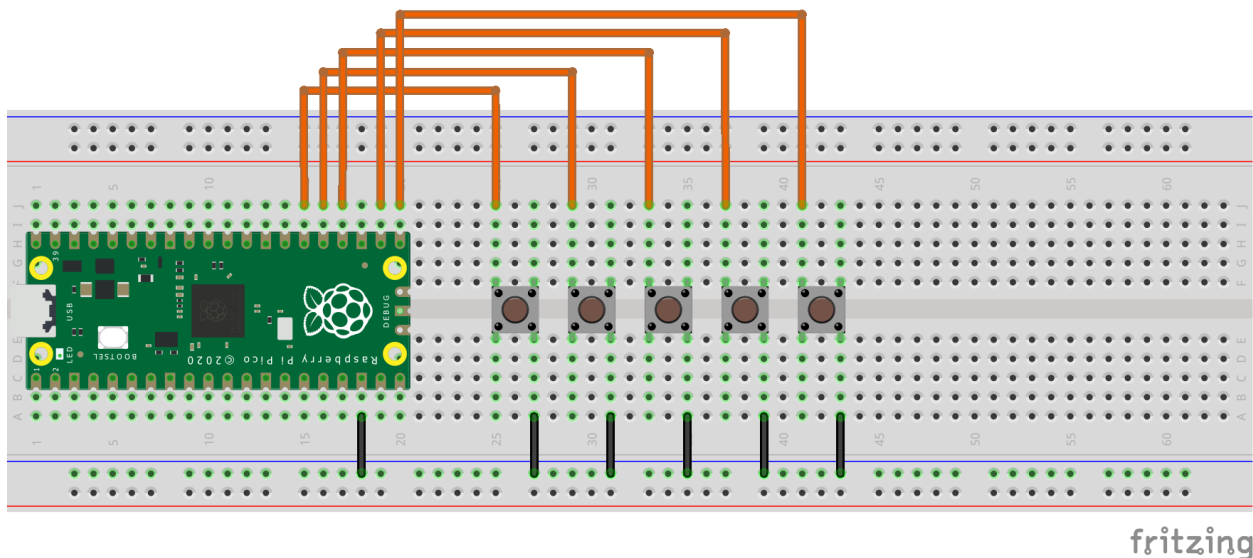
Typically OSs have a CAPS LOCK state that is global to all attached keyboards. This means that our CAPS LOCK LED will be on even if CAPS LOCK was activated by another keyboard and if we activate CAPS LOCK in our keyboard it will affect all other attached keyboards.

Lets see the steps for implementing a keyboard device.

1. In the `CMakeLists.txt`, add to the libraries `tinyusb_device` and `tiny_usb_board`
2. Include a `tusb_config.h` file. I started from the one at the `dev_hid_composite` SDK example. The important part here is setting the number of devices per class in the `CFG_TUD_defines`. You also have to define the buffer size.
3. Include a `usb_descriptors.c` file. Again, I started from the one at the `dev_hid_composite` SDK example. In this file, declare the device, report, configuration and string descriptors. You should also define here the callbacks that return these descriptors.
4. In the main file (`kbddevice.c`) initialize the USB stack by calling `board_init()` and `tusb_init()`.
5. In the main loop, call `tud_task()`. I am also calling a `hid_task()` that I wrote to do the periodic check of key presses and releases and send reports as necessary.
6. Implement a series of callback routines. The important one here is `tud_hid_set_report_cb` that is called when the host uses `SET_REPORT` to control the keyboard LEDs.

Notice that I used a dummy VID/PID (0xDEAD, 0xBEEF). **You should not use them for a device that will go “in the wild”.**

The wiring for this example is just 5 buttons connected to the Pi Pico GPIOs:



Keyboard device wiring

The checking of key presses and releases is done by the `kbd_check()` routine. It will update the keycodes to be sent in a report. `hid_task()` periodically call `kbd_check()` and send a report if necessary. The actual sending of the report is request by the `tud_hid_keyboard_report()` function of

the tinyusb library. This function receives a `report_id` (zero in our case, as we have only one device), the modifiers byte (zero as we are not implementing modifier keys like Shift) and the keycode array.

tusb_config.h (partial)

```

1  //-----
2  // DEVICE CONFIGURATION
3  //-----
4
5  #ifndef CFG_TUD_ENDPOINT0_SIZE
6  #define CFG_TUD_ENDPOINT0_SIZE    64
7  #endif
8
9  //----- CLASS -----//
10 #define CFG_TUD_HID                1
11 #define CFG_TUD_CDC                0
12 #define CFG_TUD_MSC                0
13 #define CFG_TUD_MIDI               0
14 #define CFG_TUD_VENDOR             0
15
16 // HID buffer size Should be sufficient to hold ID (if any) + Data
17 #define CFG_TUD_HID_EP_BUFSIZE     8

```

usb_descriptors.c (partial)

```

1  #include "tusb.h"
2  #include "pico/unique_id.h"
3
4  // You should use your own VID & PID !//
5  #define USBD_VID (0xDEAD)
6  #define USBD_PID (0xBEAF)
7
8  #define USBD_DESC_LEN (TUD_CONFIG_DESC_LEN + TUD_CDC_DESC_LEN)
9  #define USBD_MAX_POWER_MA (250)
10
11 #define USBD_STR_0 (0x00)
12 #define USBD_STR_MANUF (0x01)
13 #define USBD_STR_PRODUCT (0x02)
14 #define USBD_STR_SERIAL (0x03)
15
16 //-----+
17 // Device Descriptors
18 //-----+
19 static const tusb_desc_device_t usbd_desc_device = {

```

```

20     .bLength = sizeof(tusb_desc_device_t),
21     .bDescriptorType = TUSB_DESC_DEVICE,
22     .bcdUSB = 0x0200,
23     .bDeviceClass = 0x00,
24     .bDeviceSubClass = 0x00,
25     .bDeviceProtocol = 0x00,
26     .bMaxPacketSize0 = CFG_TUD_ENDPOINT0_SIZE,
27     .idVendor = USBID_VID,
28     .idProduct = USBID_PID,
29     .bcdDevice = 0x0100,
30     .iManufacturer = USBID_STR_MANUF,
31     .iProduct = USBID_STR_PRODUCT,
32     .iSerialNumber = USBID_STR_SERIAL,
33     .bNumConfigurations = 1,
34 };
35
36 //-----+
37 // HID Report Descriptor
38 //-----+
39
40 uint8_t const desc_hid_keyboard_report[] =
41 {
42     TUD_HID_REPORT_DESC_KEYBOARD()
43 };
44
45 //-----+
46 // Configuration Descriptor
47 //-----+
48
49 #define CONFIG_TOTAL_LEN  (TUD_CONFIG_DESC_LEN + TUD_HID_DESC_LEN)
50
51 #define EPNUM_KEYBOARD    0x81
52
53 uint8_t const desc_configuration[] =
54 {
55     // Config number, interface count, string index, total length, attribute, power in\
56     mA
57     TUD_CONFIG_DESCRIPTOR(1, 1, 0, CONFIG_TOTAL_LEN, TUSB_DESC_CONFIG_ATT_REMOTE_WAKEU\
58     P, 100),
59
60     // Interface number, string index, protocol, report descriptor len, EP In address,\
61     size & polling interval
62     TUD_HID_DESCRIPTOR(0, 0, HID_ITF_PROTOCOL_KEYBOARD, sizeof(desc_hid_keyboard_repor\

```

```

63 t), EPNUM_KEYBOARD, CFG_TUD_HID_EP_BUFSIZE, 10),
64 };
65
66 static char usbd_serial_str[PICO_UNIQUE_BOARD_ID_SIZE_BYTES * 2 + 1];
67
68 //-----+
69 // String Descriptors
70 //-----+
71
72 static const char *const usbd_desc_str[] = {
73     [USBD_STR_MANUF] = "Raspberry Pi",
74     [USBD_STR_PRODUCT] = "Pico",
75     [USBD_STR_SERIAL] = usbd_serial_str,
76 };
77
78
79 // Invoked when received GET DEVICE DESCRIPTOR
80 const uint8_t *tud_descriptor_device_cb(void) {
81     return (const uint8_t *)&usbd_desc_device;
82 }
83
84 // Invoked when received GET CONFIGURATION DESCRIPTOR
85 const uint8_t *tud_descriptor_configuration_cb(__unused uint8_t index) {
86     return desc_configuration;
87 }
88
89 // Invoked when received GET HID REPORT DESCRIPTOR
90 uint8_t const * tud_hid_descriptor_report_cb(uint8_t instance)
91 {
92     return desc_hid_keyboard_report;
93 }
94
95 // Invoked when received GET STRING DESCRIPTOR request
96 const uint16_t *tud_descriptor_string_cb(uint8_t index, __unused uint16_t langid) {
97     #define DESC_STR_MAX (20)
98     static uint16_t desc_str[DESC_STR_MAX];
99
100     // Assign the SN using the unique flash id
101     if (!usbd_serial_str[0]) {
102         pico_get_unique_board_id_string(usbd_serial_str, sizeof(usbd_serial_str));
103     }
104
105     uint8_t len;

```

```

106     if (index == 0) {
107         desc_str[1] = 0x0409; // supported language is English
108         len = 1;
109     } else {
110         if (index >= sizeof(usbd_desc_str) / sizeof(usbd_desc_str[0])) {
111             return NULL;
112         }
113         const char *str = usbd_desc_str[index];
114         for (len = 0; len < DESC_STR_MAX - 1 && str[len]; ++len) {
115             desc_str[1 + len] = str[len];
116         }
117     }
118
119     // first byte is length (including header), second byte is string type
120     desc_str[0] = (uint16_t) ((TUSB_DESC_STRING << 8) | (2 * len + 2));
121
122     return desc_str;
123 }

```

kbddevice.c)

```

1  /**
2   * @file kbddevice.c
3   * @author Daniel Quadros
4   * @brief A five key USB keyboard device
5   * @version 0.1
6   * @date 2022-06-21
7   *
8   * Based in the dev_hid_composite example in the Pico C SDK
9   * that is based in the tinyusb hid_boot_interface example
10  *
11  * Copyright (c) 2019 Ha Thach (tinyusb.org)
12  * @copyright Copyright (c) 2022, Daniel Quadros
13  *
14  */
15
16 #include <stdlib.h>
17 #include <stdio.h>
18 #include <string.h>
19
20 #include "bsp/board.h"
21 #include "tusb.h"
22 #include "pico/stdlib.h"

```



```

23 #include "hardware/gpio.h"
24
25 // Raspberry Pi Pico LED - Used for CAPS LOCK
26 #define LED_PIN 25
27
28 //-----+
29 // Keyboard control
30 //-----+
31
32 // Keys are connect between a pin and ground
33 uint kbd_pin[] = { 20, 19, 18, 17, 16 };
34 #define NKEYS (sizeof(kbd_pin)/sizeof(uint))
35
36 // USB codes for the keys
37 uint8_t kbd_code[] = { 0x1E, 0x1F, 0x20, 0x21, HID_KEY_CAPS_LOCK };
38
39 // Are the keys pressed?
40 bool key_pressed[NKEYS];
41 uint nkeys_pressed = 0;
42
43 // Last reported keycodes
44 uint8_t keycode[6] = { 0 };
45
46 //-----+
47 // Local routines
48 //-----+
49 void kbd_init(void);
50 void kbd_check(void);
51 void hid_task(void);
52
53 //-----+
54 // Main Program
55 //-----+
56 int main(void)
57 {
58     // Initialize the LED
59     gpio_init(LED_PIN);
60     gpio_set_dir(LED_PIN, GPIO_OUT);
61     gpio_put(LED_PIN, 0);
62
63     // Initialize the "keyboard"
64     kbd_init();
65

```

```

66 // Initialize the USB Stack
67 board_init();
68 tusb_init();
69
70 // Main loop
71 while (1)
72 {
73     tud_task();
74     hid_task();
75 }
76
77 return 0;
78 }
79
80 //-----+
81 // Keyboard
82 //-----+
83
84 // Keyboard Initialization
85 void kbd_init() {
86     for (int i = 0; i < NKEYS; i++) {
87         uint pin = kbd_pin[i];
88         gpio_init(pin);
89         gpio_set_dir(pin, GPIO_IN);
90         gpio_pull_up(pin);
91         key_pressed[i] = false;
92     }
93 }
94
95 // Check for keys pressed and released and update global variables
96 void kbd_check() {
97     for (int i = 0; i < NKEYS; i++) {
98         bool pressed = !gpio_get(kbd_pin[i]); // pressed = low
99         if (pressed != key_pressed[i]) {
100             // changed state
101             if (pressed) {
102                 // Try to put key in report
103                 for (int j = 0; j < 6; j++) {
104                     if (keycode[j] == 0) {
105                         keycode[j] = kbd_code[i];
106                         key_pressed[i] = true;
107                         nkeys_pressed++;
108                         break;

```

```

109         }
110     }
111     } else {
112         // remove from report
113         for (int j = 0; j < 6; j++) {
114             if (keycode[j] == kbd_code[i]) {
115                 keycode[j] = 0;
116                 key_pressed[i] = false;
117                 nkeys_pressed--;
118                 break;
119             }
120         }
121     }
122 }
123 }
124 }
125
126 //-----+
127 // Device callbacks
128 //-----+
129
130 // Invoked when device is mounted
131 void tud_mount_cb(void) {
132 }
133
134 // Invoked when device is unmounted
135 void tud_umount_cb(void) {
136 }
137
138
139 //-----+
140 // USB HID
141 //-----+
142
143 // Send the HID report
144 static void send_hid_report()
145 {
146     // skip if hid is not ready yet
147     if ( !tud_hid_ready() ) {
148         return;
149     }
150
151     // use to avoid send multiple consecutive zero report for keyboard

```

```
152     static bool notified_key = false;
153
154     if (nkeys_pressed) {
155         // We have keys pressed
156         tud_hid_keyboard_report(0, 0, keycode);
157         notified_key = true;
158     } else
159     {
160         // No key pressed, send empty report just one time
161         if (notified_key) {
162             tud_hid_keyboard_report(0, 0, NULL);
163             notified_key = false;
164         }
165     }
166 }
167
168 // Every 10ms, we will sent a report
169 void hid_task(void)
170 {
171     // Poll every 10ms
172     const uint32_t interval_ms = 10;
173     static uint32_t start_ms = 0;
174
175     // Check if is time for an update
176     if ( (board_millis() - start_ms) < interval_ms) {
177         return;
178     }
179     start_ms += interval_ms;
180
181     // Check the keys in the keyboard
182     kbd_check();
183
184     // Remote wakeup
185     if ( tud_suspended() && (nkeys_pressed > 0) )
186     {
187         // Wake up host if we are in suspend mode
188         // and REMOTE_WAKEUP feature is enabled by host
189         tud_remote_wakeup();
190     } else
191     {
192         send_hid_report();
193     }
194 }
```

```

195
196 // Invoked when received GET_REPORT control request
197 // Application must fill buffer report's content and return its length.
198 // Return zero will cause the stack to STALL request
199 uint16_t tud_hid_get_report_cb(uint8_t instance, uint8_t report_id, hid_report_type_t re\
200 t report_type, uint8_t* buffer, uint16_t reqlen)
201 {
202     // TODO not Implemented
203     (void) instance;
204     (void) report_id;
205     (void) report_type;
206     (void) buffer;
207     (void) reqlen;
208
209     return 0;
210 }
211
212 // Invoked when received SET_REPORT control request or
213 // received data on OUT endpoint ( Report ID = 0, Type = 0 )
214 void tud_hid_set_report_cb(uint8_t instance, uint8_t report_id, hid_report_type_t re\
215 port_type, uint8_t const* buffer, uint16_t bufsize)
216 {
217     (void) instance;
218
219     if (report_type == HID_REPORT_TYPE_OUTPUT)
220     {
221         if (bufsize) {
222             // Update the Caps Lock LED
223             gpio_put(LED_PIN, (buffer[0] & KEYBOARD_LED_CAPSLOCK)? 1 : 0);
224         }
225     }
226 }

```

Example - Connecting a PC Keyboard to the Pi Pico

Now we are going to use the Pi Pico as a host and connect a standard US QWERTY keyboard (using an OTG adapter). We will only support the boot protocol, with minimum keycode decoding. We will treat the Caps Lock key and LED, but will not implement auto repeat.

Since will be using the USB to connect the keyboard, the output will be sent to UART0. See in the UART chapter the options for connecting the UART0 to a PC and see the output.

My code is based on the `host_cdc_msc_hid` SDK example, that is itself based on the `tinysb cdc_msc_hid` example. I left only the keyboard support and enhanced it.

The steps for implementing a host that supports a keyboard device are:

1. In the `CMakeLists.txt`, add to the libraries `tinysb_host` and `tiny_usb_board`
2. Include a `tusb_config.h` file. I started from the one at the `host_cdc_msc_hid` SDK example. The important part here is setting the number of devices per class in the `CFG_TUH_` defines. You also have to define some buffers size.
4. In the main file (`kbdhost.c`) initialize the USB stack by calling `board_init()` and `tusb_init()`.
5. In the main loop, call `tuh_task()`. I am also calling a `hid_task()` that will update the keyboard LEDs.
6. Implement a series of callback routines.

To decode the keycodes I am using the `HID_KEYCODE_TO_ASCII` table that is in `tinysb`. As mentioned, this table is valid for the standard US QWERTY keyboard.

`tusb_config.h` (partial)

```

1  //-----
2  // CONFIGURATION
3  //-----
4
5  // Size of buffer to hold descriptors and other data used for enumeration
6  #define CFG_TUH_ENUMERATION_BUFSIZE 256
7
8  #define CFG_TUH_HUB          1
9  #define CFG_TUH_CDC          0
10 #define CFG_TUH_HID          4 // typical keyboard + mouse device can have 3\
11 -4 HID interfaces
12 #define CFG_TUH_MSC          0
13 #define CFG_TUH_VENDOR       0
14
15 #define CFG_TUSB_HOST_DEVICE_MAX (CFG_TUH_HUB ? 5 : 1) // normal hub has 4 ports
16
17 //----- HID -----//
18
19 #define CFG_TUH_HID_EP_BUFSIZE 64
20 #define CFG_TUH_HID_EPOUT_BUFSIZE 64

```

kbdhost.c

```

1  /**
2   * @file kbdhost.c
3   * @author Daniel Quadros
4   * @brief A USB keyboard host
5   * @version 0.1
6   * @date 2022-06-21
7   *
8   * Based in the host_cdc_msc_hid example in the Pico C SDK
9   * that is based in the tinyusb cdc_msc_hid example
10  *
11  * Copyright (c) 2019 Ha Thach (tinyusb.org)
12  * @copyright Copyright (c) 2022, Daniel Quadros
13  *
14  */
15
16 #include <stdlib.h>
17 #include <stdio.h>
18 #include <string.h>
19
20 #include "bsp/board.h"
21 #include "tusb.h"
22 #include "pico/stdlib.h"
23 #include "hardware/uart.h"
24
25 // Select UART and Pins
26 #define UART_ID uart0
27 #define UART_TX_PIN 0
28 #define UART_RX_PIN 1
29
30 // keycodes translation table
31 #define NKEYS 128
32 static uint8_t const keycode2ascii[NKEYS][2] = {HID_KEYCODE_TO_ASCII};
33
34 #define MAX_KEY 6 // Maximun number of pressed key in the boot layout report
35
36 // Caps lock control
37 static bool capslock_key_down_in_last_report = false;
38 static bool capslock_key_down_in_this_report = false;
39 static bool capslock_on = false;
40
41 // Keyboard LED control
42 static uint8_t leds = 0;

```

```

43 static uint8_t prev_leds = 0xFF;
44
45 // Keyboard address and instance (assumes there is only one)
46 static uint8_t keybd_dev_addr = 0xFF;
47 static uint8_t keybd_instance;
48
49 // Each HID instance has multiple reports
50 #define MAX_REPORT 4
51 static uint8_t _report_count[CFG_TUH_HID];
52 static tuh_hid_report_info_t _report_info_arr[CFG_TUH_HID][MAX_REPORT];
53
54 //-----+
55 // Local routines
56 //-----+
57 void serial_init(void);
58 void hid_task(void);
59 static void process_kbd_report(hid_keyboard_report_t const *report);
60
61 //-----+
62 // Main Program
63 //-----+
64 int main(void)
65 {
66     // Initialize the UART
67     serial_init();
68
69     // Initialize the USB Stack
70     board_init();
71     tusb_init();
72
73     // Main loop
74     while (1)
75     {
76         tuh_task();
77         hid_task();
78     }
79
80     return 0;
81 }
82
83 //-----+
84 // UART Initialization
85 //-----+

```



```

86 void serial_init()
87 {
88     // Set up UART, parameters will be overwritten later
89     uart_init(UART_ID, 115200);
90     uart_set_hw_flow(UART_ID, false, false);
91     uart_set_format(UART_ID, 8, 1, UART_PARITY_NONE);
92     uart_set_fifo_enabled(UART_ID, false);
93
94     // Set the TX and RX pins
95     gpio_set_function(UART_TX_PIN, GPIO_FUNC_UART);
96     gpio_set_function(UART_RX_PIN, GPIO_FUNC_UART);
97 }
98
99 //-----+
100 // This will be called by the main loop
101 //-----+
102 void hid_task(void)
103 {
104     // update keyboard leds
105     if (keybd_dev_addr != 0xFF)
106     { // only if keyboard attached
107         if (leds != prev_leds)
108         {
109             tuh_hid_set_report(keybd_dev_addr, keybd_instance, 0, HID_REPORT_TYPE_OUTPUT, \
110 &leds, sizeof(leds));
111             prev_leds = leds;
112         }
113     }
114 }
115
116 //-----+
117 // TinyUSB Callbacks
118 //-----+
119
120 // Invoked when device with hid interface is mounted
121 void tuh_hid_mount_cb(uint8_t dev_addr, uint8_t instance, uint8_t const *desc_report\
122 , uint16_t desc_len)
123 {
124     // Report descriptor is also available for use. tuh_hid_parse_report_descriptor()
125     // can be used to parse common/simple enough descriptor.
126     _report_count[instance] = tuh_hid_parse_report_descriptor(_report_info_arr[instanc\
127 e], MAX_REPORT, desc_report, desc_len);
128     // Check if at least one of the reports is for a keyboard

```

```

129     for (int i = 0; i < _report_count[instance]; i++)
130     {
131         if ((_report_info_arr[instance][i].usage_page == HID_USAGE_PAGE_DESKTOP) &&
132             (_report_info_arr[instance][i].usage == HID_USAGE_DESKTOP_KEYBOARD))
133         {
134             keybd_dev_addr = dev_addr;
135             keybd_instance = instance;
136         }
137     }
138
139     // request to receive report
140     tuh_hid_receive_report(dev_addr, instance);
141 }
142
143 // Invoked when device with hid interface is un-mounted
144 void tuh_hid_umount_cb(uint8_t dev_addr, uint8_t instance)
145 {
146     keybd_dev_addr = 0xFF; // keyboard not available
147 }
148
149 // Invoked when received report from device via interrupt endpoint
150 void tuh_hid_report_received_cb(uint8_t dev_addr, uint8_t instance, uint8_t const *r\
151 eport, uint16_t len)
152 {
153     uint8_t const rpt_count = _report_count[instance];
154     tuh_hid_report_info_t *rpt_info_arr = _report_info_arr[instance];
155     tuh_hid_report_info_t *rpt_info = NULL;
156
157     if ((rpt_count == 1) && (rpt_info_arr[0].report_id == 0))
158     {
159         // Simple report without report ID as 1st byte
160         rpt_info = &rpt_info_arr[0];
161     }
162     else
163     {
164         // Composite report, 1st byte is report ID, data starts from 2nd byte
165         uint8_t const rpt_id = report[0];
166
167         // Find report id in the array
168         for (uint8_t i = 0; i < rpt_count; i++)
169         {
170             if (rpt_id == rpt_info_arr[i].report_id)
171                 {

```

```

172     rpt_info = &rpt_info_arr[i];
173     break;
174 }
175 }
176
177 report++;
178 len--;
179 }
180
181 if (rpt_info && (rpt_info->usage_page == HID_USAGE_PAGE_DESKTOP))
182 {
183     switch (rpt_info->usage)
184     {
185     case HID_USAGE_DESKTOP_KEYBOARD:
186         // Assume keyboard follow boot report layout
187         process_kbd_report((hid_keyboard_report_t const *)report);
188         break;
189
190     default:
191         break;
192     }
193 }
194
195 // continue to request to receive report
196 tuh_hid_receive_report(dev_addr, instance);
197 }
198
199 //-----+
200 // Keyboard
201 //-----+
202
203 // look up key in a report
204 static inline bool find_key_in_report(hid_keyboard_report_t const *report, uint8_t k\
205 eycode)
206 {
207     for (uint8_t i = 0; i < MAX_KEY; i++)
208     {
209         if (report->keycode[i] == keycode)
210         {
211             return true;
212         }
213     }
214 }

```

```

215     return false;
216 }
217
218 // process keyboard report
219 static void process_kbd_report(hid_keyboard_report_t const *report)
220 {
221     static hid_keyboard_report_t prev_report = {0, 0, {0}}; // previous report to check
222     key released
223
224     // Check caps lock
225     capslock_key_down_in_this_report = find_key_in_report(report, HID_KEY_CAPS_LOCK);
226     if (capslock_key_down_in_this_report && !capslock_key_down_in_last_report)
227     {
228         // CAPS LOCK was pressed
229         capslock_on = !capslock_on;
230         if (capslock_on)
231         {
232             leds |= KEYBOARD_LED_CAPSLOCK;
233         }
234         else
235         {
236             leds &= ~KEYBOARD_LED_CAPSLOCK;
237         }
238     }
239
240     // check other pressed keys
241     for (uint8_t i = 0; i < MAX_KEY; i++)
242     {
243         uint8_t key = report->keycode[i];
244         if ((key != 0) && (key != HID_KEY_CAPS_LOCK) && !find_key_in_report(&prev_report, key))
245         { // ignore fillers, Caps lock and keys already pressed
246             // Find corresponding ASCII code
247             uint8_t ch = (key < NKEYS) ? keycode2ascii[key][0] : 0; // unshifted key code,
248             to test for letters
249             bool const is_ctrl = report->modifier & (KEYBOARD_MODIFIER_LEFTCTRL | KEYBOARD_
250             _MODIFIER_RIGHTCTRL);
251             bool is_shift = report->modifier & (KEYBOARD_MODIFIER_LEFTSHIFT | KEYBOARD_MOD
252             IFIER_RIGHTSHIFT);
253             if (capslock_on && (ch >= 'a') && (ch <= 'z'))
254             {
255                 // capslock affects only letters
256                 is_shift = !is_shift;

```

```

258     }
259     ch = (key < NKEYS) ? keycode2ascii[key][is_shift ? 1 : 0] : 0;
260     if (is_ctrl)
261     {
262         // control char
263         if ((ch >= 0x60) && (ch <= 0x7F))
264         {
265             ch = ch - 0x60;
266         }
267         else if ((ch >= 0x40) && (ch <= 0x5F))
268         {
269             ch = ch - 0x40;
270         }
271     }
272
273     if (ch)
274     {
275         // send key code to UART
276         uart_putc_raw(UART_ID, ch);
277     }
278 }
279 }
280
281 // save current status
282 prev_report = *report;
283 capslock_key_down_in_last_report = capslock_key_down_in_this_report;
284 }

```

Example - Serial USB Adapter

I will not delve into all the details of this example, but I am including it because it can be useful. Most modern PCs no longer have a RS232 serial interface, asynchronous serial communication must be done through an adapter that implements the CDC USB class.

In this example I use the tinyusb library to implement a CDC device that will send data received from the UART0 to the PC and send through the UART0 data received from the PC.

Like previous device examples, this requires:

- setting up configuration in `tusb_config.h`
- creating and initializing descriptors in `usb_descriptors.c`
- calling `board_init()` and `tusb_init()` at the beginning of the application

- calling `tud_task()` in the main loop
- implement a series of callbacks

One thing that may confuse you is `tinyusb` support for multiple CDC ports. There are two sets of functions, one with names starting with `tud_cdc_n_` and another with names starting with `tud_cdc_`. The former has an extra parameter, `itf` to select the port, the latter assume the first port (`itf = 0`). The callbacks will always pass the `itf` parameter. In my example there is only one port, so I use the `tud_cdc_` functions and ignore the `itf` parameter in the callbacks.

A concern here is the PC driver for our CDC device. Linux and Windows 10 provide drivers for generic CDC devices, so you can just plug to the PC. Previous versions of Windows look for an INF file, based on the VID/PID, so our device will not work.

In this example I am using a dummy VID/PID (0xDEAD, 0xBEEF). **You should not use them for a device that will go “in the wild”.**

The functions used for controlling the UART are described in the respective chapter.

`tusb_config.h` (partial)

```

1  //-----
2  // DEVICE CONFIGURATION
3  //-----
4
5  #ifndef CFG_TUD_ENDPOINT0_SIZE
6  #define CFG_TUD_ENDPOINT0_SIZE    64
7  #endif
8
9  //----- CLASS -----//
10 #define CFG_TUD_HID                0
11 #define CFG_TUD_CDC                1
12 #define CFG_TUD_MSC                0
13 #define CFG_TUD_MIDI               0
14 #define CFG_TUD_VENDOR             0
15
16 // CDC FIFO size of TX and RX
17 #define CFG_TUD_CDC_RX_BUFSIZE     (TUD_OPT_HIGH_SPEED ? 512 : 64)
18 #define CFG_TUD_CDC_TX_BUFSIZE     (TUD_OPT_HIGH_SPEED ? 512 : 64)
19
20 // CDC Endpoint transfer buffer size, more is faster
21 #define CFG_TUD_CDC_EP_BUFSIZE     (TUD_OPT_HIGH_SPEED ? 512 : 64)

```

usb_descriptors.c (partial)

```

1  #include "tusb.h"
2  #include "pico/unique_id.h"
3
4  // You should use your own VID & PID !
5  #define USBD_VID (0xDEAD)
6  #define USBD_PID (0xBEAF)
7
8  #define USBD_DESC_LEN (TUD_CONFIG_DESC_LEN + TUD_CDC_DESC_LEN)
9  #define USBD_MAX_POWER_MA (250)
10
11 #define USBD_ITF_CDC      (0) // needs 2 interfaces
12 #define USBD_ITF_MAX      (2)
13
14 #define USBD_CDC_EP_CMD (0x81)
15 #define USBD_CDC_EP_OUT (0x02)
16 #define USBD_CDC_EP_IN (0x82)
17 #define USBD_CDC_CMD_MAX_SIZE (8)
18 #define USBD_CDC_IN_OUT_MAX_SIZE (64)
19
20 #define USBD_STR_0 (0x00)
21 #define USBD_STR_MANUF (0x01)
22 #define USBD_STR_PRODUCT (0x02)
23 #define USBD_STR_SERIAL (0x03)
24 #define USBD_STR_CDC (0x04)
25
26 // Note: descriptors returned from callbacks must exist long enough for transfer to \
27 complete
28
29 static const tusb_desc_device_t usbd_desc_device = {
30     .bLength = sizeof(tusb_desc_device_t),
31     .bDescriptorType = TUSB_DESC_DEVICE,
32     .bcdUSB = 0x0200,
33     .bDeviceClass = TUSB_CLASS_MISC,
34     .bDeviceSubClass = MISC_SUBCLASS_COMMON,
35     .bDeviceProtocol = MISC_PROTOCOL_IAD,
36     .bMaxPacketSize0 = CFG_TUD_ENDPOINT0_SIZE,
37     .idVendor = USBD_VID,
38     .idProduct = USBD_PID,
39     .bcdDevice = 0x0100,
40     .iManufacturer = USBD_STR_MANUF,
41     .iProduct = USBD_STR_PRODUCT,
42     .iSerialNumber = USBD_STR_SERIAL,

```

```

43     .bNumConfigurations = 1,
44 };
45
46 static const uint8_t usbd_desc_cfg[USBD_DESC_LEN] = {
47     TUD_CONFIG_DESCRIPTOR(1, USBD_ITF_MAX, USBD_STR_0, USBD_DESC_LEN,
48         0, USBD_MAX_POWER_MA),
49
50     TUD_CDC_DESCRIPTOR(USBD_ITF_CDC, USBD_STR_CDC, USBD_CDC_EP_CMD,
51         USBD_CDC_CMD_MAX_SIZE, USBD_CDC_EP_OUT, USBD_CDC_EP_IN, USBD_CDC_IN_OUT_MAX_
52 SIZE),
53
54 };
55
56 static char usbd_serial_str[PICO_UNIQUE_BOARD_ID_SIZE_BYTES * 2 + 1];
57
58 static const char *const usbd_desc_str[] = {
59     [USBD_STR_MANUF] = "Raspberry Pi",
60     [USBD_STR_PRODUCT] = "Pico",
61     [USBD_STR_SERIAL] = usbd_serial_str,
62     [USBD_STR_CDC] = "CDC Example",
63 };
64
65 const uint8_t *tud_descriptor_device_cb(void) {
66     return (const uint8_t *)&usbd_desc_device;
67 }
68
69 const uint8_t *tud_descriptor_configuration_cb(__unused uint8_t index) {
70     return usbd_desc_cfg;
71 }
72
73 const uint16_t *tud_descriptor_string_cb(uint8_t index, __unused uint16_t langid) {
74     #define DESC_STR_MAX (20)
75     static uint16_t desc_str[DESC_STR_MAX];
76
77     // Assign the SN using the unique flash id
78     if (!usbd_serial_str[0]) {
79         pico_get_unique_board_id_string(usbd_serial_str, sizeof(usbd_serial_str));
80     }
81
82     uint8_t len;
83     if (index == 0) {
84         desc_str[1] = 0x0409; // supported language is English
85         len = 1;

```



```

86     } else {
87         if (index >= sizeof(usbd_desc_str) / sizeof(usbd_desc_str[0])) {
88             return NULL;
89         }
90         const char *str = usbd_desc_str[index];
91         for (len = 0; len < DESC_STR_MAX - 1 && str[len]; ++len) {
92             desc_str[1 + len] = str[len];
93         }
94     }
95
96     // first byte is length (including header), second byte is string type
97     desc_str[0] = (uint16_t) ((TUSB_DESC_STRING << 8) | (2 * len + 2));
98
99     return desc_str;
100 }

```

usbserial.c

```

1  /**
2   * @file usbserial.c
3   * @author Daniel Quadros
4   * @brief Example of a simple USB Serial Adapter
5   * @version 0.1
6   * @date 2022-06-20
7   *
8   * @copyright Copyright (c) 2022, Daniel Quadros
9   *
10  */
11
12  #include <stdlib.h>
13  #include <stdio.h>
14  #include <string.h>
15
16  #include "bsp/board.h"
17  #include "tusb.h"
18  #include "pico/stdlib.h"
19  #include "hardware/uart.h"
20
21  // Select UART and Pins
22  #define UART_ID uart0
23  #define UART_TX_PIN 0
24  #define UART_RX_PIN 1
25

```

```
26 // Raspberry Pi Pico LED
27 #define LED_PIN 25
28
29 // Local routines
30 void serial_init(void);
31 void cdc_task(void);
32
33 // Main Program
34 int main(void)
35 {
36     // Initialize the LED
37     gpio_init(LED_PIN);
38     gpio_set_dir(LED_PIN, GPIO_OUT);
39     gpio_put(LED_PIN, 0);
40
41     // Initialize the UART
42     serial_init();
43
44     // Initialize the USB Stack
45     board_init();
46     tusb_init();
47
48     // Main loop
49     while (1)
50     {
51         tud_task();
52         cdc_task();
53     }
54
55     return 0;
56 }
57
58 // UART Initialization
59 void serial_init() {
60     // Set up UART, parameters will be overwritten later
61     uart_init(UART_ID, 115200);
62     uart_set_hw_flow(UART_ID, false, false);
63     uart_set_format(UART_ID, 8, 1, UART_PARITY_NONE);
64     uart_set_fifo_enabled(UART_ID, true);
65
66     // Set the TX and RX pins
67     gpio_set_function(UART_TX_PIN, GPIO_FUNC_UART);
68     gpio_set_function(UART_RX_PIN, GPIO_FUNC_UART);
```

```

69  }
70
71
72  //-----+
73  // Device callbacks
74  //-----+
75
76  // Invoked when device is mounted
77  void tud_mount_cb(void) {
78  }
79
80  // Invoked when device is unmounted
81  void tud_umount_cb(void) {
82  }
83
84
85  //-----+
86  // USB CDC
87  //-----+
88
89  // Moves data between USB and UART
90  // Not optimized!
91  void cdc_task(void) {
92      // connected() check for DTR bit, its assume that the application
93      // in the host set it when connecting
94      if ( tud_cdc_connected() ) {
95
96          // send trough the USB data received by the UART
97          if (uart_is_readable(UART_ID)) {
98              while (uart_is_readable(UART_ID) && (tud_cdc_write_available() > 0)) {
99                  tud_cdc_write_char(uart_getc(UART_ID));
100              }
101              tud_cdc_write_flush(); // so we don't wait for a full buffer to send
102          }
103
104          // send trough the UART data received by the USB
105          while (uart_is_writable(UART_ID) && (tud_cdc_available() > 0)) {
106              uart_putc_raw(UART_ID, tud_cdc_read_char());
107          }
108      } else {
109          // ignore data received through the UART
110          while (uart_is_readable(UART_ID)) {
111              uart_getc(UART_ID);

```

```
112     }
113
114     // ignore data received through the USB
115     if (tud_cdc_available() > 0) {
116         tud_cdc_read_flush();
117     }
118 }
119 }
120
121 // Invoked when cdc when line state changed e.g connected/disconnected
122 void tud_cdc_line_state_cb(uint8_t itf, bool dtr, bool rts) {
123     (void) itf;
124     (void) rts;
125
126     // TODO set some indicator
127     if ( dtr )
128     {
129         // Terminal connected
130         gpio_put(LED_PIN, 1);
131     } else
132     {
133         // Terminal disconnected
134         gpio_put(LED_PIN, 0);
135     }
136 }
137
138 // Invoked when line coding is change via SET_LINE_CODING
139 void tud_cdc_line_coding_cb(uint8_t itf, cdc_line_coding_t const* p_line_coding) {
140
141     // 0: 1 stop bit - 1: 1.5 stop bits - 2: 2 stop bits
142     uint stop_bits = 2;
143     if (p_line_coding->stop_bits == 0) {
144         stop_bits = 1;
145     }
146
147     // 0: None - 1: Odd - 2: Even - 3: Mark - 4: Space
148     // TODO: implement Mark & Space parity
149     uart_parity_t parity = UART_PARITY_NONE;
150     if (p_line_coding->parity == 1) {
151         parity = UART_PARITY_ODD;
152     } else if (p_line_coding->parity == 2) {
153         parity = UART_PARITY_EVEN;
154     }
```

```
155
156     uart_set_baudrate(UART_ID, p_line_coding->bit_rate);
157     uart_set_format(UART_ID, p_line_coding->data_bits, stop_bits, parity);
158 }
```

Conclusion

And so we get to the end of this journey through the features of the RP2040. And what a journey it was!

Along the way we saw many characteristics that can help to implement our projects and enhance them:

- Dual ARM Cortex M0+ cores
- Sophisticate DMA controller
- Flexible clock generation
- Good set of peripherals: Timer, Watchdog, RTC, PWM, UART, I²C, SPI and ADC
- USB controller with host and device support
- Programmable I/O (PIO) for efficiently implementation of digital I/O

It is clear that the creators of the RP2040 have given a lot of thought to performance. Not just a high clock rate, but the ability to do many things at the same time. I have a feeling that they also included some features just for the fun!

The C/C++ SDK has an enormous number of functions, allowing (in most cases) full control of the hardware without meddling with registers and bits.

Now it is up to you, oh adventurous reader, to use all this power to create elegant and efficient projects. And, also important, have fun.

Daniel Quadros
Sept 2022

Appendix A - CMake Files for RP2040 Programs

The Raspberry Pi Pico C/C++ SDK uses CMake to create the make files that control the building of programs.

CMake can be intimidating, we are going to look here only the minimum needed for compiling RP2040 programs and generating the uf2 files that are use to load them in flash memory.

The file we need to create for each program is the “CMakeLists.txt”.

A typical file (from an example in chapter 8) is shown bellow

```
1  cmake_minimum_required(VERSION 3.13)
2
3  include(pico_sdk_import.cmake)
4
5  project(hcsr04_project)
6
7  pico_sdk_init()
8
9  add_executable(hcsr04
10     hcsr04.c
11 )
12
13 pico_generate_pio_header(hcsr04 ${CMAKE_CURRENT_LIST_DIR}/hcsr04.pio)
14
15 target_link_libraries(hcsr04 PRIVATE
16     pico_stdlib
17     pico_stdio
18     hardware_pio
19 )
20
21 pico_enable_stdio_usb(hcsr04 1)
22 pico_enable_stdio_uart(hcsr04 0)
23
24 pico_add_extra_outputs(hcsr04)
```

Lets look at each line and see what they do and what you need to change for your own file.

```
cmake_minimum_required(VERSION 3.13)
```

This line states the minimum version of CMAKE that can be used. This line can be omitted, if included try to keep it in sync with the minimum version required by the SDK.

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

Includes the contents of the `pico_sdk_import.cmake`. This file contains the definitions needed to use the SDK and must be copied to your work directory from the root directory of the SDK. Sometimes you will need other includes (for example for the definitions in `pico-extra`).

```
project(hcsr04_project)
```

This defines the name of the project, `hcsr04` in this case. The name of the project appears in many other lines, a common error is forgetting to change it on some line when editing an existing `CMakeLists.txt` for a new project.

```
pico_sdk_init()
```

This must be included after the project name definition and before the next lines, as it sets up things for using the SDK.

```
1 add_executable(hcsr04
2     hcsr04.c
3 )
```

Here you list the C files that will be compiled. The first `hcsr04` is the name of the project. You can add the name of as many files as needed, separating them by whitespace (spaces, tabs and/or newlines).

```
pico_generate_pio_header(hcsr04 ${CMAKE_CURRENT_LIST_DIR}/hcsr04.pio)
```

You will only need this line if your project includes PIO code. PIO code is written in a special format and converted into a C header file by this line. The first `hcsr04` is the name of the project, `hcsr04.pio` is the name of the file with the PIO code. You can have multiple lines like this if you have multiple PIO programs.

```
1 target_link_libraries(hcsr04 PRIVATE
2     pico_stdlib
3     pico_stdio
4     hardware_pio
5 )
```

Here the libraries used are listed. The first `hcsr04` is the name of the project.

```
1 pico_enable_stdio_usb(hcsr04 1)
2 pico_enable_stdio_uart(hcsr04 0)
```


Use this lines to control to where the `stdio` messages will be sent (see Appendix B). If you are not using `stdio` you can ommit them. `hcsr04` is the name of the project, 1 will enable and 0 disable.

```
pico_add_extra_outputs(hcsr04)
```

This enables extra outputs for the build, including the `ef2` file. Again, `hcsr04` is the name of the project.

There are a lot more that can be done with CMake, but this should be enough for your RP2040 projects. A full description of CMake can be found at <https://cmake.org/documentation/>.

Appendix B - Using stdio

Even with the availability of advanced debuggers, developers still use `printf()` to display debugging messages. The RP2040 C/C++ SDK has limited support for the standard input output (stdio) routines. The standard input and output can be used through USB or UART.

To use stdio in your project:

- include the library `pico_stdlib` in `CMakeLists.txt`
- enable stdio on USB or UART in `CMakeLists.txt`
- call `stdio_init_all()` or `'stdio_usb_init()` or one of the `stdxx_uart_init` functions in your application initialization. `stdio_init_all()` will initialize USB and/or UART (depending on `CMakeLists.txt`) with default values. `stdio_uart_init_full` allows to initialize stdio on UART with full control of the parameters.

Enabling stdio in CMakeLists.txt

```
1  cmake_minimum_required(VERSION 3.13)
2
3  include(pico_sdk_import.cmake)
4
5  project(myproj_project)
6
7  pico_sdk_init()
8
9  add_executable(myproj
10     main.c
11 )
12
13 target_link_libraries(myproj PRIVATE
14     pico_stdlib
15 )
16
17 pico_enable_stdio_usb(myproj 1)
18 pico_enable_stdio_uart(myproj 0)
19
20 pico_add_extra_outputs(myproj)
```

The '1' in `pico_enable_stdio_usb(myproj 1)` enables stdio through USB, the '0' in `pico_enable_stdio_uart(myproj 0)` disables stdio through the UART.

See in the UART example some information about how to connect the RP2040 UART through a PC using a serial USB adapter. One of the PIO examples is using a Raspberry Pi Pico as a minimum serial USB adapter.

The default configuration of the UART for the Pico board is:

```
1 #define PICO_DEFAULT_UART 0
2 #define PICO_DEFAULT_UART_TX_PIN 0
3 #ifndef PICO_DEFAULT_UART_RX_PIN
4 #define PICO_DEFAULT_UART_BAUD_RATE 115200
```

When using stdio to communicate to a PC through USB, the RP2040 will appear as an USB CDC device, accessible as serial port. Depending on the PC operating system a driver may be necessary. While using the USB can be more practical (as no adapter is needed) it has a significant binary size cost.

Selected pico_stdio Functions

```
void stdio_init_all (void)
```

Initializes stdio on USB and/or UART, based on the settings in CMakeLists.txt. If the UART is initialized, default configuration is used.

```
int getchar_timeout_us (uint32_t timeout_us)
```

Get a character from stdin if there is one available within timeout microseconds.

Returns the character (0 to 255) or PICO_ERROR_TIMEOUT.

```
int putchar_raw (int c)
```

Sends a character through stdout with no conversions.

```
int puts_raw (const char *s)
```

Sends a string through stdout with no conversions.

Selected pico_stdio_uart Functions

```
void stdout_uart_init (void)
```

This function initialize the UART with the default configuration for standard output only.

```
void stdin_uart_init (void)
```

This function initialize the UART with the default configuration for standard input only.

```
void stdio_uart_init_full (uart_inst_t *uart, uint baud_rate, int tx_pin, int rx_pin)
```

This function initialize the UART with an specific configuration and assign it for standard input and output.

Selected pico_stdio_usb Functions

`bool stdio_usb_init (void)`

Initializes USB for standard input and output.

`bool stdio_usb_connected (void)`

Returns true if a CDC is established through the USB. This means not only that the USB is connected and recognized, but also that some software has opened the corresponding serial port.

This function is useful to make sure you will not lose messages sent before you start your communication program in the PC. It is used in many of my examples.

The printf Function

The `printf` function in the SDK is a lightweight version by Marco Paland (official repository is at <https://github.com/mpaland/printf>).

The `printf` function has the following prototype:

```
printf(const char *format, ...)
```

where `...` means zero or more parameters of any type.

The format string contains the text to be printed with optional *format specifiers* embedded. The format specifiers determine how the parameters are printed. Association between format specifiers and parameters is made left to right.

Here is a typical example of using `printf`:

```
1 int counter = 5;
2 unsigned mask = 42;
3 printf ("Counter = %d, mask = %04X\n", counter, mask);
```

The `%d` will be replaced by the decimal representation of the content of `counter` and `%04X` will be replaced by the hexadecimal representation of the content of `mask` with four digits (leading 0 added as needed). The `\n` is a newline character and will be converted to carriage return + line feed characters. The output will be:

```
Counter = 5, mask = 002A
```

The information below was extracted from the README.md file of the project, (c) Marco Paland (info@paland.com) 2014-2019, PALANDesign Hannover, Germany - MIT License.

Format Specifiers

A format specifier follows this prototype: `%[flags][width][.precision][length]type`

Supported Types

Type	Output
d or i	Signed decimal integer
u	Unsigned decimal integer
b	Unsigned binary
o	Unsigned octal
x	Unsigned hexadecimal integer (lowercase)
X	Unsigned hexadecimal integer (uppercase)
f or F	Decimal floating point
e or E	Scientific-notation (exponential) floating point
g or G	Scientific or decimal floating point
c	Single character
s	String of characters
p	Pointer address
%	A % followed by another % character will write a single %

Supported Flags

Flags	Description
-	Left-justify within the given field width; Right justification is the default.
+	Forces to precede the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a - sign.
(space)	If no sign is going to be written, a blank space is inserted before the value.
#	Used with o, b, x or X specifiers the value is preceded with 0, 0b, 0x or 0X respectively for values different than zero. Used with f, F it forces the written output to contain a decimal point even if no more digits follow. By default, if no digits follow, no decimal point is written.
0	Left-pads the number with zeros (0) instead of spaces when padding is specified (see width sub-specifier).

Supported Width

Width	Description
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The width is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

Supported Precision

Precision	Description
.number	For integer specifiers (d, i, o, u, x, X): precision specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A precision of 0 means that no character is written for the value 0. For f and F specifiers: this is the number of digits to be printed after the decimal point. By default, this is 6, maximum is 9. For s: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. If the period is specified without an explicit value for precision, 0 is assumed.
*	The precision is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

Supported Length

The length sub-specifier modifies the length of the data type.

Length	d i	u o x X
(none)	int	unsigned int
hh	char	unsigned char
h	short int	unsigned short int
l	long int	unsigned long int
ll	long long int	unsigned long long int
j	intmax_t	uintmax_t
z	size_t	size_t
t	ptrdiff_t	ptrdiff_t

Appendix C - Debugging Using the SWD Port

Debugging embedded projects requires some way to remotely interfere with normal program execution and access the microcontroller registers.

The RP2040 includes support for ARM's SWD (Serial Wire Debug). This is a two wire (SWDCLK and SWDIO) serial interface. The Raspberry Pi Pico (and most RP2040 boards) have a connector with this two signals plus ground.

To connect the SWD port to a PC we need some intelligent device that can implement the SWD protocol at one end and talk to a debugger through USB at the other. The Raspberry Pi Foundation answer to this is... a firmware for the Raspberry Pi Pico (called **picoprobe**).

As a bonus, this firmware also sets up a CDC device. This is useful if the RP2040 you are debugging (the *target*) is using the USB port. You can send debug messages through one of the UARTs of the target and connect it to a UART of the *debugger* Pico (the one with the picoprobe software).

The full instructions for setting up the picoprobe and the PC debugger are in Appendix A of the official “[Getting started with Raspberry Pi Pico](#)”¹ document. Here I will highlight a few important points.

Picoprobe Connections

By default, the picoprobe software uses this pins in the *debugger* Pico:

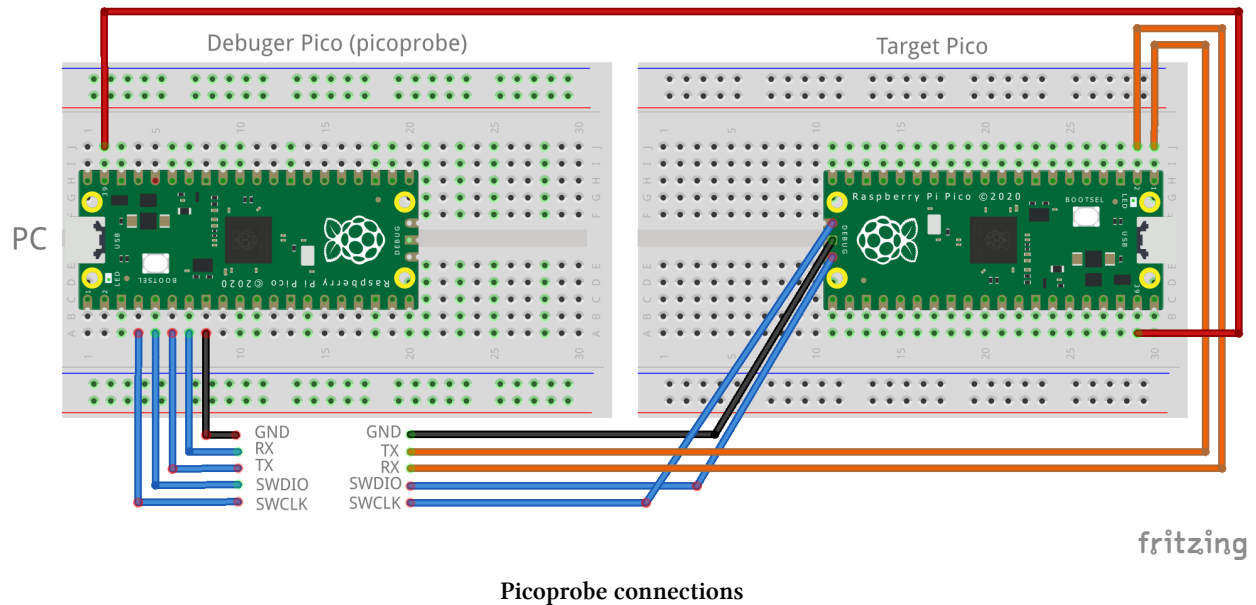
Pin	Function
GP2	SWCLK
GP3	SWDIO
GP4	UART TX
GP5	UART RX

As just four pins are used, you may consider using a small factor RP2040 board instead of a Pico. The picoprobe firmware is provided in source code, so you can change this pins if not appropriate for your board (in `picoprobe_config.h`).

At a minimum you will have to connect SWCLK, SWDIO and GND between the debugger and the target. If you want to connect the UART, notice that the TX pin of one side should be connected to the RX pin of the other side.

¹<https://datasheets.raspberrypi.com/pico/getting-started-with-pico.pdf>

The debugger Pico will be powered by the USB connection to the PC. Depending on the power requirements of the target circuit, you may power it by connecting the VSYS of the two Picos, as long as the target is not powered by other sources.



Software Installation

Besides installing the picoprobe firmware in the debugger Pico and interconnecting the two Picos, you will need to:

- Build, from the source code, a version of the OpenOCD software with the picoprobe driver enabled. Full instructions for Linux, Windows and MacOS can be found in the Getting Started document.
- If you are using Windows in your PC you will need a driver for the SWD interface. Again, full instructions for downloading and installing can be found in the Getting Started document.
- To use the CDC interface you will need a serial communication program.

Building OpenOCD (specially under Windows) is a long process. As an alternative you can find an executable version at

<https://github.com/earlephilhower/pico-quick-toolchain/releases/>

The usual caution on downloading and running executables from the Internet applies. Earle F. Philhower, III is the responsible for the unofficial Raspberry Pi Pico Arduino core

Debugging from the Command Line

To start a debugging session you will:

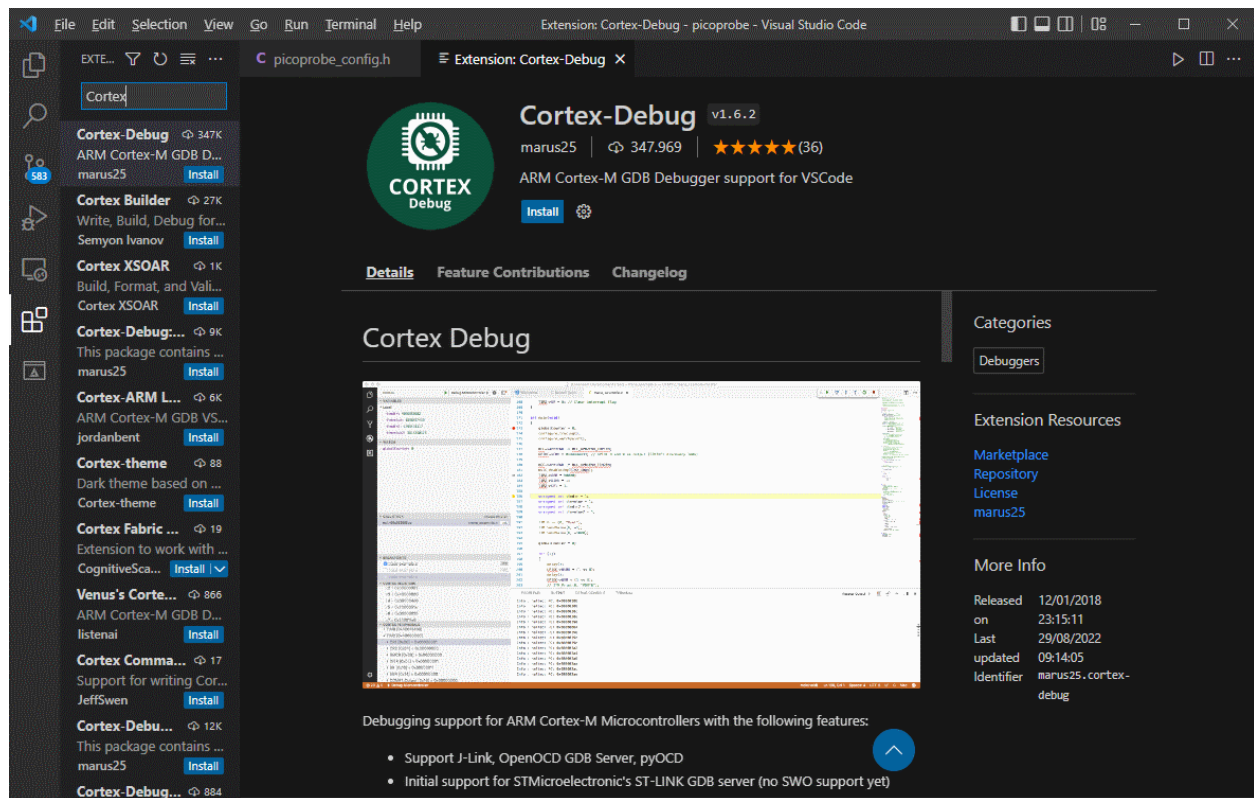
- start OpenOCD using the command `src/openocd -f interface/picoprobe.cfg -f target/rp2040.cfg -s tcl`
- start the gdc debugger that is installed along the compiler when you set up the SDK
- connect GDB to the target with the command `target remote localhost:3333`

You can find tutorials and cheat sheets for GDB in the Internet.

Debugging from inside Visual Code

While this takes some work to set up, once it is done you can debug from inside Visual Code and not worry about command line commands.

The first thing you need to debug inside Visual Studio is install the Cortex-Debug extension.



Cortex-Debug extension installation

To setup up debugging in a project, you need to create two configuration files inside a subdirectory of your project named `.vscode`. Once you created these files you can copy them to any project you want to debug.

The first configuration file is name `launch.json` and its contents should be as follows:

```

1  {
2      "version": "0.2.0",
3      "configurations": [
4          {
5              "name": "Pico Debug",
6              "cwd": "${workspaceRoot}",
7              "executable": "${command:cmake.launchTargetPath}",
8              "request": "launch",
9              "type": "cortex-debug",
10             "serverType": "openocd",
11             "gdbPath": "arm-none-eabi-gdb",
12             "device": "RP2040",
13             "configFiles": [
14                 "interface/picoprobe.cfg",
15                 "target/rp2040.cfg"
16             ],
17             "svdFile": "${env:PICO_SDK_PATH}/src/rp2040/hardware_regs/rp2040.svd",
18             "runToMain": true,
19             // Work around for stopping at main on restart
20             "postRestartCommands": [
21                 "break main",
22                 "continue"
23             ],
24             "searchDir": ["D:/msys64/home/Daniel/openocd/tcl"],
25         }
26     ]
27 }
```

The value for `searchDir` is the path to the `tcl` subdirectory of where you put OpenOCD. In my case, I built OpenOCD from the source and installed MSYS2 at `D:\msys64` and cloned the OpenOCD repository into my “home” directory, resulting the path `D:/msys64/home/Daniel/openocd/tcl`. Notice I am using forward slashes, if you use backslashes you need to duplicate them.

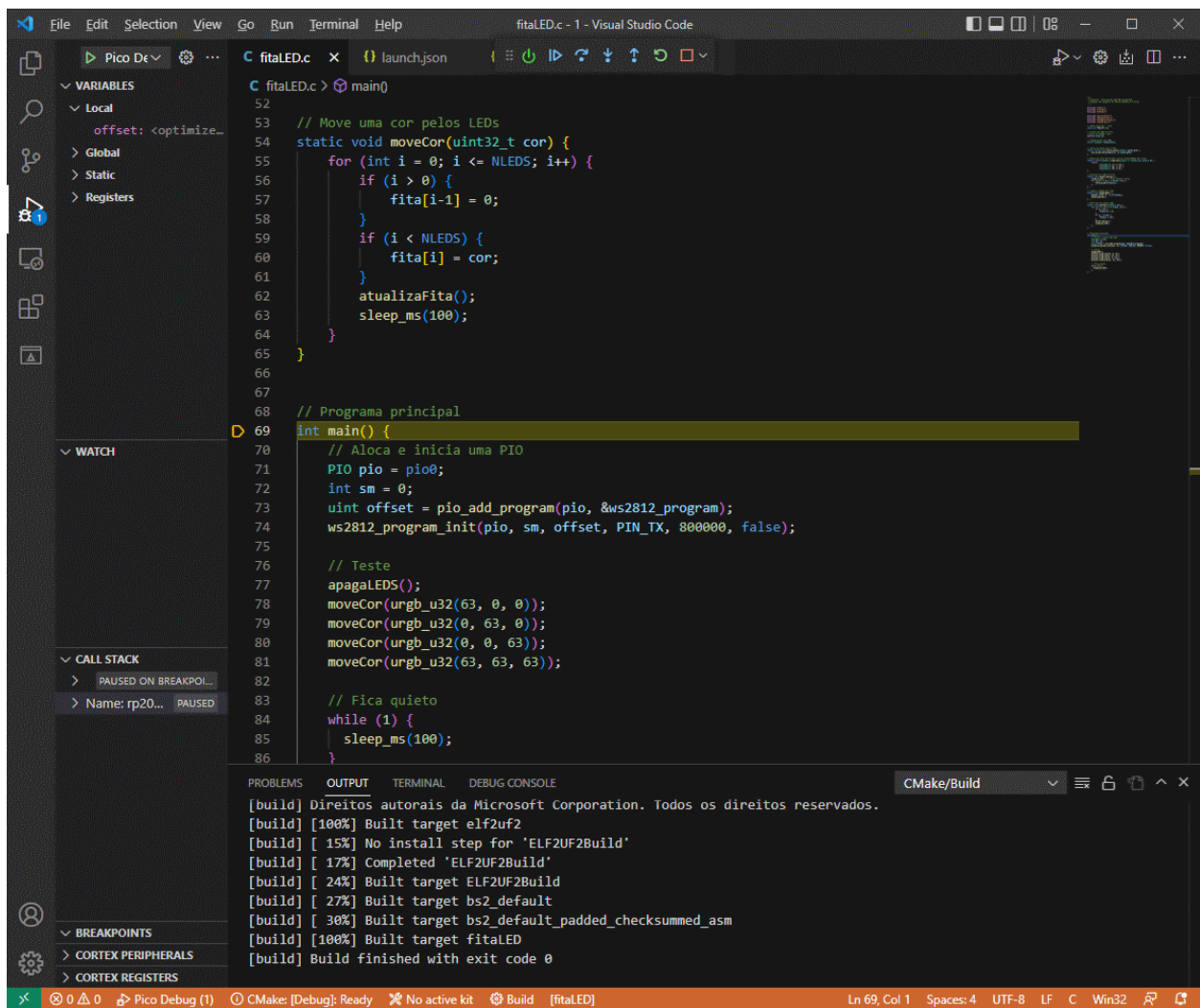
The second file is `settings.json` and it should have:

```
1 {
2     // These settings tweaks to the cmake plugin will ensure
3     // that you debug using cortex-debug instead of trying to launch
4     // a Pico binary on the host
5     "cmake.statusbar.advanced": {
6         "debug": {
7             "visibility": "hidden"
8         },
9         "launch": {
10             "visibility": "hidden"
11         },
12         "build": {
13             "visibility": "default"
14         },
15         "buildTarget": {
16             "visibility": "hidden"
17         }
18     },
19     "cmake.buildBeforeRun": true,
20     "C_Cpp.default.configurationProvider": "ms-vscode.cmake-tools",
21     "cortex-debug.openocdPath": "D:/msys64/home/Daniel/openocd/src/openocd.exe"
22 }
```

Change `cortex-debug.openocdPath` to the path to the `openocd` executable (`openocd.exe` under Windows).

If you have built OpenOCD from source under Windows, copy the `libusb-1.0.dll` file from `msys64/mingw64/bin` to the same directory as `openocd.exe` to make sure Windows will find it.

Now you can build your project, install it on the target RP2040 and debug by just pressing F5 (or selecting Run and Debug in the icons in the left and pressing the green triangle besides Pico Debug).



Debugging in Visual Code

Appendix D - Accessing the RP2040 Registers

The ARM processors interact with the hardware built into the RP2040 through **registers**. All registers are memory mapped, they have memory addresses associated to them.

For most tasks you can leave to the SDK functions to manipulate the RP2040 registers. In this appendix we will look how we can directly access the registers and talk about the efficiency of the SDK functions for GPIO output.

Registers Addresses and Basic Access

The addresses and bit functions for the registers are documented in the RP2040 datasheet. In the C/C++ SDK they are defined in various include files under `src/rp2040/hardware_regs`. To make the addresses definitions more “IDE friendly” and simplify its access, structures are defined in the include files under `src/rp2040/hardware_structs`.

Let’s take a look at some of the definitions for the ADC to understand how this works. In `adc.h` under `hardware_structs` we have:

```
1  typedef struct {
2      _REG_(ADC_CS_OFFSET) // ADC_CS
3      io_rw_32 cs;
4
5      _REG_(ADC_RESULT_OFFSET) // ADC_RESULT
6      io_ro_32 result;
7
8      _REG_(ADC_FCS_OFFSET) // ADC_FCS
9      io_rw_32 fcs;
10
11     _REG_(ADC_FIFO_OFFSET) // ADC_FIFO
12     io_ro_32 fifo;
13
14     _REG_(ADC_DIV_OFFSET) // ADC_DIV
15     io_rw_32 div;
16
17     _REG_(ADC_INTR_OFFSET) // ADC_INTR
18     io_ro_32 intr;
```

```

19
20     _REG_(ADC_INTE_OFFSET) // ADC_INTE
21     io_rw_32 inte;
22
23     _REG_(ADC_INTF_OFFSET) // ADC_INTF
24     io_rw_32 intf;
25
26     _REG_(ADC_INTS_OFFSET) // ADC_INTS
27     io_ro_32 ints;
28 } adc_hw_t;
29
30 #define adc_hw ((adc_hw_t *)ADC_BASE)

```

Starting at the end, `adc_hw` will access the structure `adc_hw_t` at the address `ADC_BASE` (that is defined in the `addressmap.h` under `hardware_regs` as `0x4004c000`).

The `_REG_()` macro expands to nothing, it is there so that the IDE can find the names of the offsets.

To inform the compiler that the registers can change from outside the code (they are *volatile*) and indicate that some of them are read-only or write-only, a few typedefs are used:

```

1 typedef volatile uint32_t io_rw_32;
2 typedef const volatile uint32_t io_ro_32;
3 typedef volatile uint32_t io_wo_32;

```

To access a register you just have to deference a pointer:

```

1 // Check if the most recent ADC conversion encountered an error
2 bool adc_error = adc_hw->cs & ADC_CS_ERR_BITS;
3 // Disable ADC interrupts
4 adc_hw->inte = 0;

```

Special Write Operations

What we saw in the previous section might look sufficient for all register operations... until you start to worry about concurrency.

To see why, let's look at digital output. The state of the GPIO pins is controlled by register `SIO_GPIO_OUT`, each bit in this register controls a GPIO pin. Suppose we want to turn on (set to HIGH) `GPIO0`. We could do this:

```
1 sio_hw->gpio_out |= 0x01;
```

This will compile into something like these ARM instructions:

```
1 movs r4, #208      ; 0xd0
2 lsls r4, r4, #24    ; R4 = 0xd0000000 = SIO_BASE
3 ldr r3, [r4, #16] ; R3 = content of SIO_GPIO_OUT
4 movs r2, #1
5 orrs r3, r2        ; R3 = R# | 1
6 str r3, [r4, #16] ; SIO_GPIO_OUT = R3
```

This is OK, as long as there is no one else who can change SIO_GPIO_OUT between the time we read and the time we write it back. If we also want to change any pin in an interrupt handler, we got a problem. If the interrupt occurs between the ldr and str instructions, the change made by the interrupt handler will be overwritten.

We can disable interrupts before changing SIO_GPIO_OUT and re-enable them after the change. This is tedious and error prone. Worse, it won't help if code on the other core also changes SIO_GPIO_OUT.

What we really need is to change a register in an **atomic** way (that is, in an operation that cannot be broken in parts). This is available in the RP2040 through different addresses for the SIO registers. Using these addresses will access the same register but perform different operations when writing.

Using the 0xd0000000 SIO_BASE addresses will just write the written value to the addressed register. There are three more variations:

- Writes to the 0xd0001000 region will perform a **XOR** operation: the register will be updated to the XOR of its current value and the written value (that is, bits with value 1 in the written value will invert the corresponding bit in the register).
- Writes to the 0xd0002000 region will perform a **SET** operation: the register will be updated to the OR of its current value and the written value.
- Writes to the 0xd0003000 region will perform a **CLR** operation: the register will be updated to the AND of its current value and the complement of the written value (that is, bits with value 1 in the written value will force a 0 in the register).

The SDK has defines for the address modifiers, macros to generate a modified address and, most important, inline functions for executing this operations:

```
1 hw_set_bits(io_rw_32 *addr, uint32_t mask);
2 hw_clear_bits(io_rw_32 *addr, uint32_t mask);
3 hw_xor_bits(io_rw_32 *addr, uint32_t mask);
```

Going back to our example of turning on (set to HIGH) GPIO0, we can write:

```
1 hw_set_bits(sio_hw->gpio_out, 0x01);
```

The update of the `SIO_GPIO_OUT` will be done in a single `str` instruction. And be atomic, with no risk of concurrency problems.

In the particular case of GPIOs we can also write this as:

```
1 sio_hw->gpio_set = 0x01;
```

Using the SDK Functions for GPIO Output

TL;DR:: Just use the SDK functions and don't worry.

The basic GPIO functions are defined as *inline* functions (they compiler will put their code where they are used instead of executing a subroutine call) in `rp2_common/hardware_gpio/include/hardware/gpio.h`. Here are some of them:

```
1 static inline void gpio_set_mask(uint32_t mask) {
2     sio_hw->gpio_set = mask;
3 }
4
5 static inline void gpio_clr_mask(uint32_t mask) {
6     sio_hw->gpio_clr = mask;
7 }
8
9 static inline void gpio_put(uint gpio, bool value) {
10     uint32_t mask = 1ul << gpio;
11     if (value)
12         gpio_set_mask(mask);
13     else
14         gpio_clr_mask(mask);
15 }
16
17 static inline void gpio_put_masked(uint32_t mask, uint32_t value) {
18     sio_hw->gpio_togl = (sio_hw->gpio_out ^ value) & mask;
19 }
```

`gpio_set_mask()` and `gpio_clr_mask()` will be compiled in just one instruction. Changes through `gpio_put` will be atomic; the compiler is smart enough to simplify the code if `gpio` or `value` are constants by computing the mask at compile time or eliminating the `if`.

If you want to change more than one pin you can use the `gpio_put_masked()` function. It changes (*toggles*) only the bits that are different from what we want. This function will generate multiple

instructions but is still concurrency-safe, *as long the other core or interrupts do not change the same pins at the same time*. If you have two pieces of code that can change the same pins at the same time, you have a logic problem, not a concurrency problem!