Micro Controller Programming

An introduction to the basics of microcontroller architecture and programming

Kacie Beckett

2024-11-21

Contents

1. MCU Overview

1.1. What is a processor?

Microprocessor (MPU) Central Processing Unit optionally with some cache (memory) in the same package. Less powerful than processors that run operating systems like Windows/MacOS/Linux.

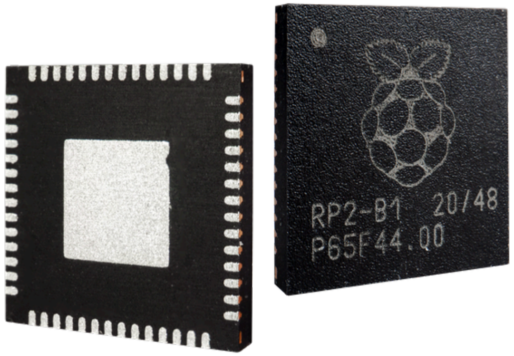
Microcontroller (MCU) Contains a microprocessor + peripherals + memory + storage in a single package. The processor is constrained and lower in performance.

System on Chip (SOC) Similar to a microcontroller but with more a powerful processor. Mobile devices are shifting to using this as opposed to a seperate processor and other elements on the same board.

Note: This is a simplification, and there can be overlap!

1.2. Introduction to the RP2040

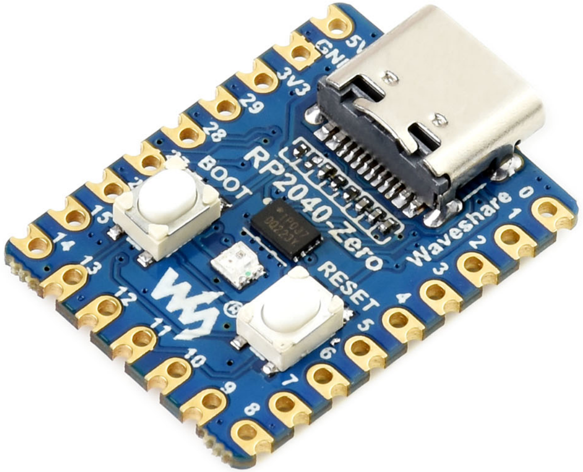
For the purposes of today's workshop we will be considering the Raspberry Pi Foundations, RP2040 Microcontroller.



- High Level Concepts and programming techniques covered today will generalise to any microcontroller

1.2. Introduction to the RP2040

We will be using the Waveshare RP2040-Zero development board which connects the micro-controller to a bunch of useful components. We will program it with our more powerful computer!



This board can be used directly with Arduino IDE just like an arduino uno board which you are likely familar with.

1.3. How does Arduino IDE Work?

- Has an abstracted top layer (your arduino code)
- Secretly adds a bunch of code for your specfic microcontroller to make it work
- Gets converted to the binary code $\{0, 1\}^{\mathbb{N}}$ which then gets uploaded
- Now it runs on the microcontroller

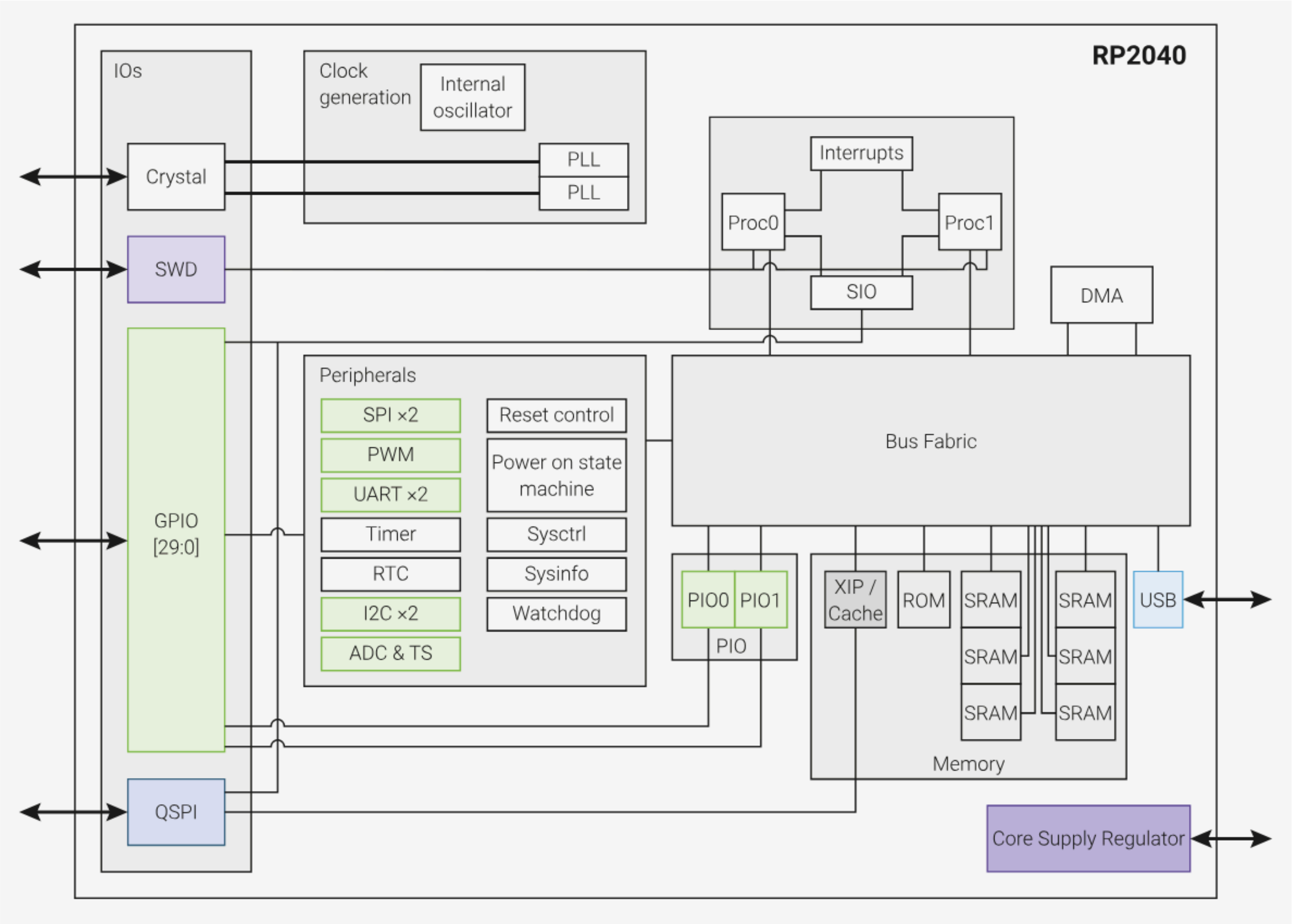
What is that secret code being run?

Its a big mix of arduino's code and the microcontroller manufacturer's code!

- Later we will take a look at the RP2040′s manufacturer code also known as the Standard Developer Kit (SDK).

2. MCU Architecture

2.1. Block Diagram

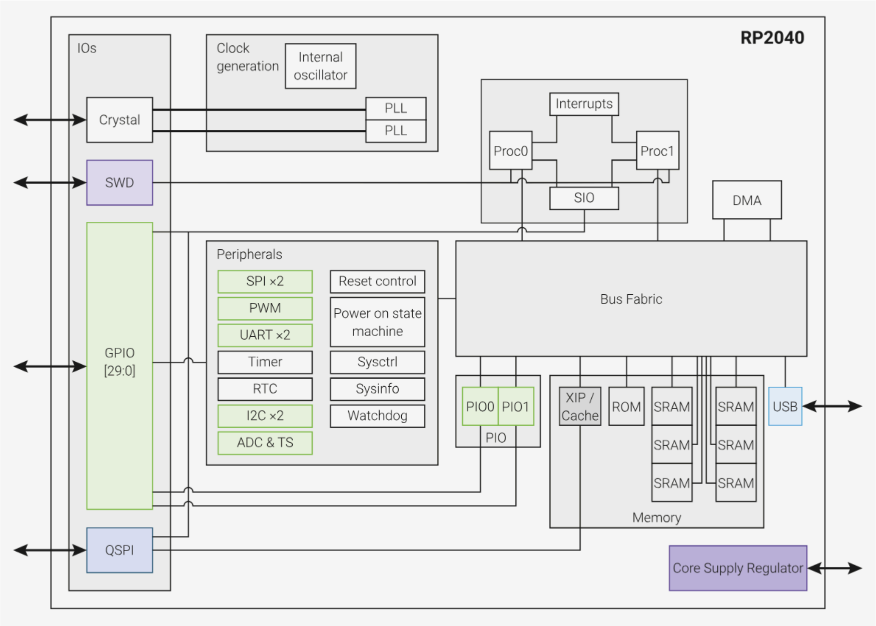


Page 10: https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf

2.1. Block Diagram

Main takeaway is we have seperate parts of the microcontroller to handle:

- the main code execution
- timers
- interupts
- general purpose input/output
- communication with peripherals
- reading and writing to memory



Page 10: https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf

Note: "Bus Fabric" contains the two microprocessor cores!

2.2. Processor Cores

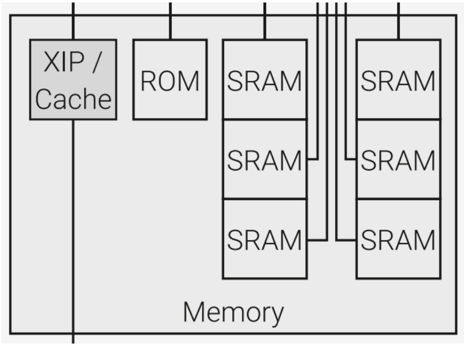
In computing, a ‘Core’ refers to an individual processing unit which is responsible for executing instructions and performing calculations.

- This is what ‘runs’ your code and controls the rest of the microcontroller

The RP2040 has **two** microprocessor cores.

2.3. Memory

- **Read Only Memory (ROM)**
  - Initial startup routine,useful libraries ect
  - Programmed at the factory
  - Non-Volatile (stores data without power)
- **Static Random Access Memory (SRAM)**
  - Used for the processor to store stuff
  - Volatile (data lost without power)
- **Flash Memory (external)**
  - Stores your program or other data accessed through execute in place (XIP) so processor can run the program directly. Has a cache to speed up access.



Extract from Section 2.1

RP2040-Zero Board Flash Chip

2.4. Registers

Term for a specific address (location) in memory which stores some specific data. Lots of stuff can have registers!

- Addressed are denoted in hexa-decimal
- Example could be a register to store if an LED is on or off
- Can be found in datasheet for specific parts

2.4.8. List of Registers

The ARM Cortex-M0+ registers start at a base address of `0xe0000000` (defined as PPB_BASE in SDK).

Offset	Name	Info
0xe010	SYST_CSR	SysTick Control and Status Register
0xe014	SYST_RVR	SysTick Reload Value Register

Page 77: https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf

2.5. Interupts

The microcontroller is complicated, so how do we make sure the microprocessor can handle all its different jobs?

- Interupts can tell the processor it needs to do something before other things!
- Interupts have priorities:
  - Higher priority tasks can interupt lower priority tasks

Necessary for timing critical tasks like communication.

2.6. Direct Memory Access (DMA)

- The DMA control gets set up via registers that control things like the source address, destination address, transfer size, source and destination address increments, total number of transfers to make, and the trigger source.
- Peripherals generally have an option to produce a DMA request instead of an interrupt request, so the CPU can spend more time executing other code.
- If a DMA channel is waiting on that trigger, it executes the configured transfer.
- When a transfer is complete you have the option to have the DMA controller generate an interrupt so the CPU can then take action on the completed transfer.

2.7. Watchdog

A special timer that can reboot the Microcontroller if it times out.

- Need to call a function to refresh the timer.
- If we are in an infinite loop we won't refresh the watchdog timer so the Microcontroller will reboot!
- Useful to prevent being the microcontroller from being stuck when we want our system to always work.

2.8. Multicore Programming

When we have multiple cores we need to be careful!

- If both microprocessors try to do an operation on the same memory/ logic unit ect, we could have an issue with overwriting each other or similar!
- Another common issue is race conditions, ie where we have non deterministic behaviour because the two cores are “racing” to achieve some task first!

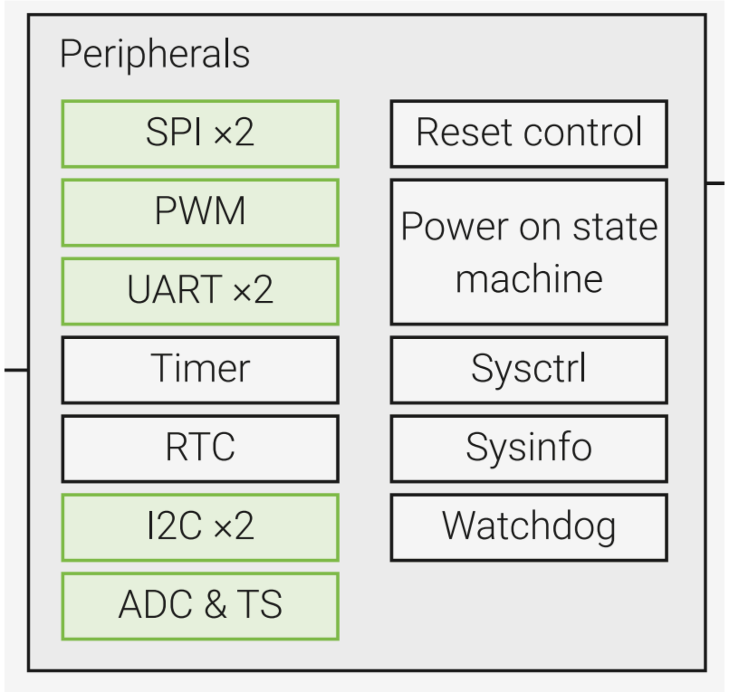
Mutexes A mutual exclusion is an execution lock. Only one core/ thread can hold the lock and it stops other cores from doing operations.

Semaphores Superset of mutex that allows n cores/threads to access/ do something with a resource.

2.9. Peripherals

We have Special Parts of MCU for handling stuff like communication (SPI, I2C, UART, PWM) and timers ect.

- In theory the processor core could do all the work but it would be hard to run other stuff and maintain accurate timing!
- Emulating communication protocols on the processor without using dedicated hardware is called **bit banging**



Extract from Section 2.1

3. Communication Protocols

3.1. UART

`Universal Asynchronous Receive Transmit` uses 3 Wires:

- RX (Receive)
- TX (Transmit)
- Ground

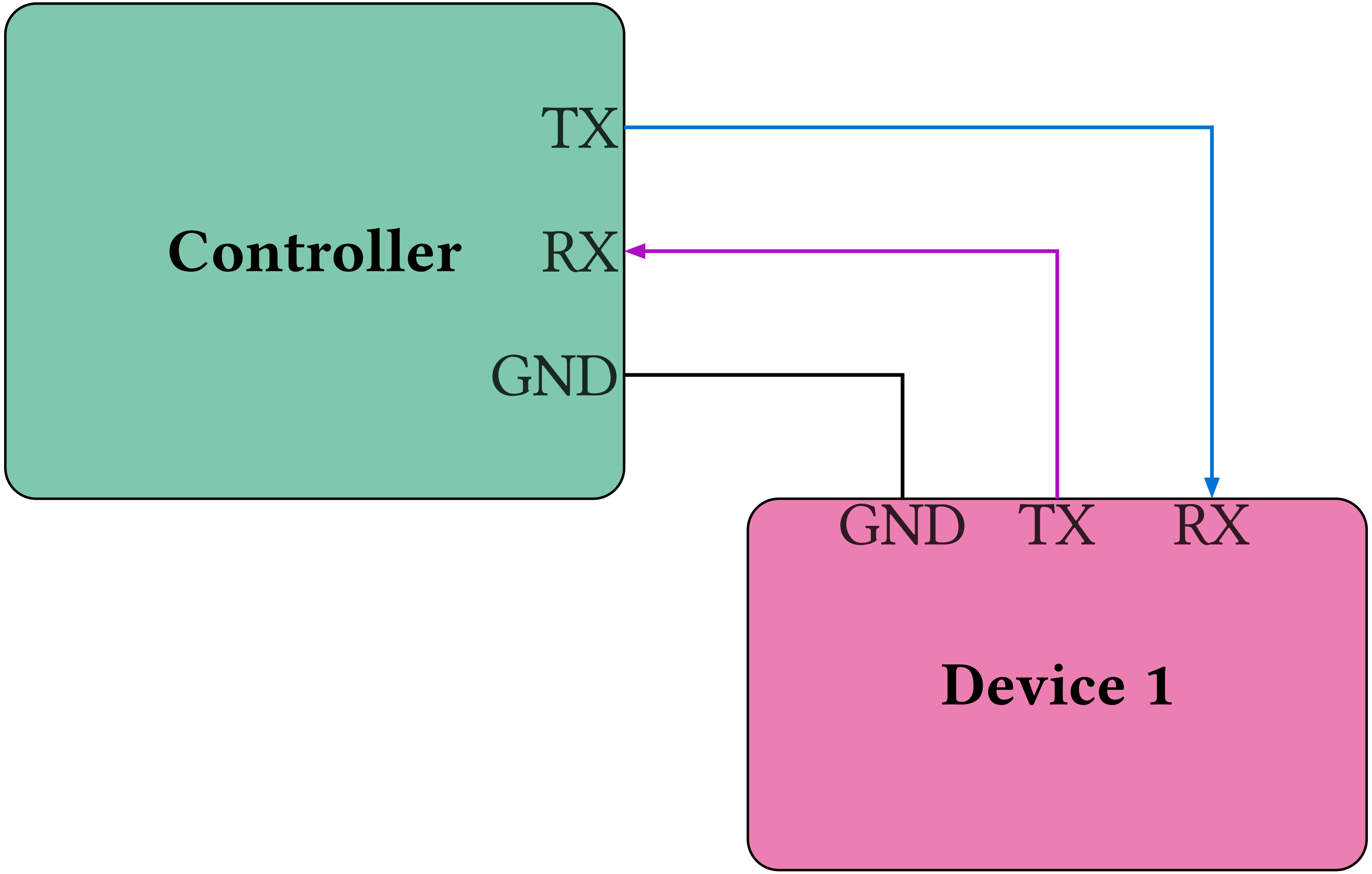
Point to point connection, does not support multiple devices!

- UART is the slowest (9600 - 115200 bits per second)
- Very simple protocol just read the incoming data

Can only do a single device to device connection. UART is also not strictly a specific protocol, but technically describes a class of communication protocols.

- Version 1.0/2.0 of USB use 2 wires and are thus are a UART protocol.

3.1. UART



3.2. I2C

`Inter-Integrated Circuit` uses 3 Wires:

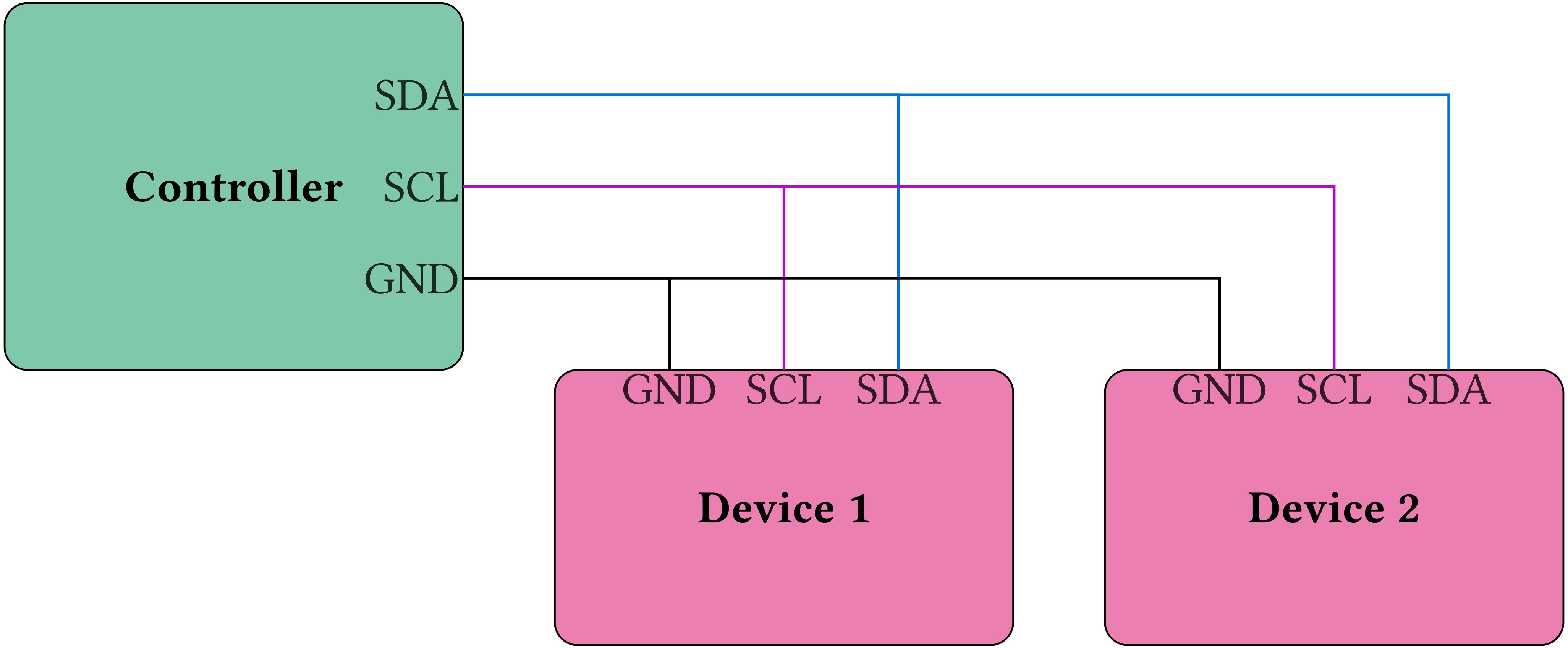
- SDA (Serial Data)
- SCL (Serial Clock)
- Ground

Can support multiple devices on the same 2 wire “bus” if they have different addresses (upto 128 per bus):

- Some devices have only one address so you need a multiplexer to use multiple on the same bus otherwise might have a jumper on the PCB.

I2C is faster than UART (100kHz default up to 3.4MHz in high speed mode) but more complicated. To use I2C you specify the device address and register you want to read data from!

3.2. I2C


Controller
SDA
SCL
GND
GND SCL SDA
Device 1
GND SCL SDA
Device 2

3.3. SPI

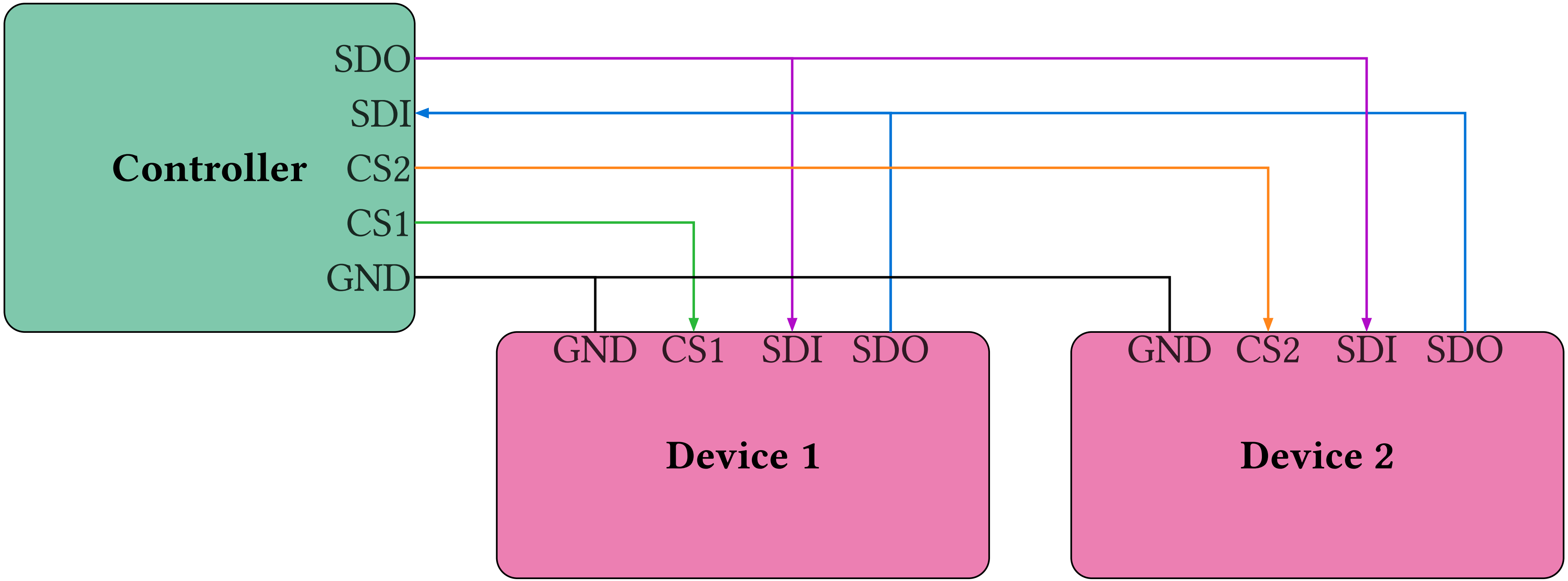
Uses atleast 4+N Wires:

- SDO/MOSI (Serial Data Out)
- SDI/MISO (Serial Data In)
- SCLK
- Ground
- N Chip Select (CS) Wires.

SPI is the faster than I2C and UART (up to 10MHz) but is more complicated and uses more pins.

- Can support less devices as each device needs a dedicated pin

3.3. SPI


Controller
SDO
SDI
CS2
CS1
GND
GND
CS1
SDI
SDO
Device 1
GND
CS2
SDI
SDO
Device 2

4. Real Time Operating Systems

4.1. Overview

The main feature of Real Time Operating Systems is that we have predictable behaviour:

- We make use of priority based interupts and a scheduler to ensure the system operates as expected
- The scheduler organises what will execute and when for us!
- Would get complicated if we tried to manage large amounts of interupts ect!

Real time means the timing is very accurate. Regular operating systems don’t necessarily make guarantees of how long a task will take.

4.2. Example: Button Press

Naive Approach:

```c
int main() {
  // main loop
  while(true) {
    if (read(button_pin) == HIGH){
      // do something
    }
    some_slow_code();
  }
}
```

Interrupt Approach:

```c
int main() {
  register_irq(button_irq,button_pin);
  // main loop
  while(true) {
    some_slow_code();
  }
}
void button_irq() {
  // do something
}
```

The interupt approach would react to the button press faster.

Note: This is a sketch not real code!

4.3. Example: LED Blinking

Naive Approach:

```c
int main() {
  // main loop
  while(true) {
    write(LED_PIN, HIGH);
    delay(1000);
    write(LED_PIN,LOW);
    delay(1000);
    printf("hello world");
  }
}
```

Scheduler Approach:

```c
void led_task() {
  write(LED_PIN, HIGH);
  virtual_task_delay(1000);
  write(LED_PIN,LOW);
  virtual_task_delay(1000);
}

int main() {
  register_task(led_task,button_pin);
  start_task_scheduler();
  // main loop
  while(true)
    printf("hello world");
}
```

We can print out hello world much more often with our scheduler!

Note: This is a sketch not real code!

4.4. FreeRTOS

An example of a Real Time Operating System is Free RTOS. This is effectively a library that we can import into our code. We still need to use the manufactures SDK.

We can then use the library functions to control scheduling and execution of tasks we write code for.

https://www.freertos.org/

4.5. Arduino

The Arduino Framework is secretly a real time operating system!

- It uses FreeRTOS under the hood

You may have had problems when trying to use multiple libraries on Arduino Framework with it not working properly.

- This is because some libraries use interupts, and these could conflict!

4.6. Zephyr OS

This is similar to the philosophy of the Arduino Framework.

Pros:

- Aims to let you write code that can be used on many microcontrollers because of heavy abstractions

Cons:

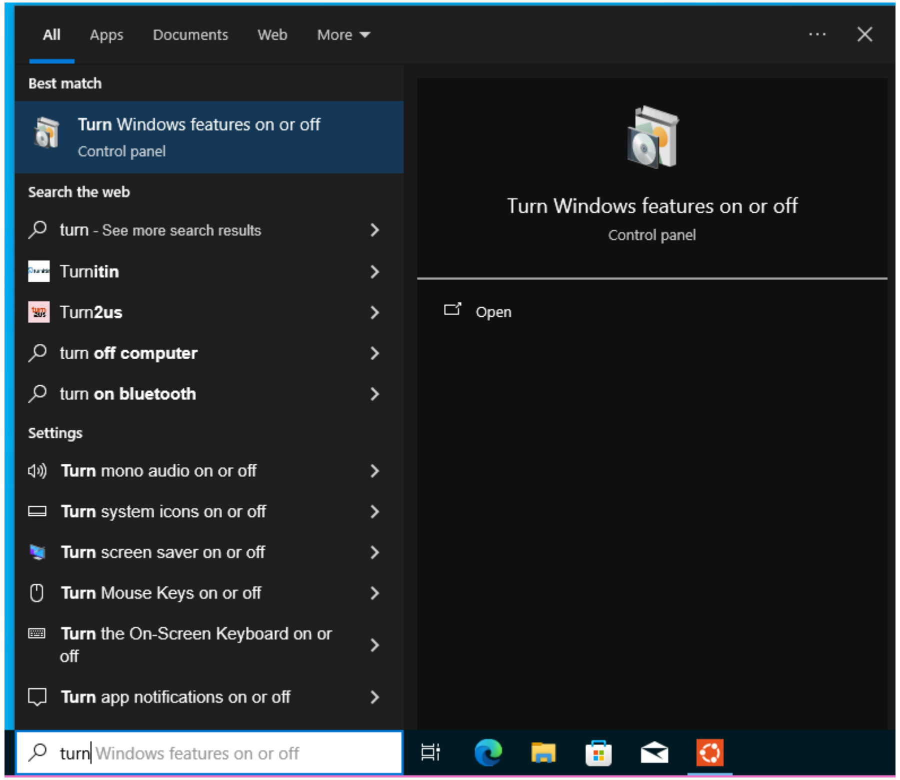
- Much more complicated to use compared to FreeRTOS because of the abstractions

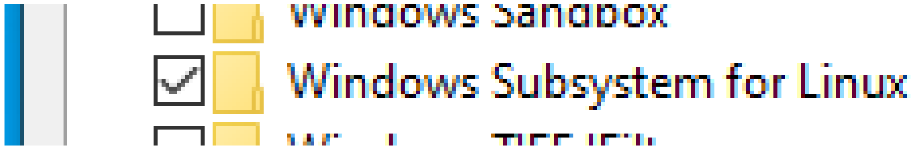
https://zephyrproject.org/

5. Software Setup

5.1. Setup Windows

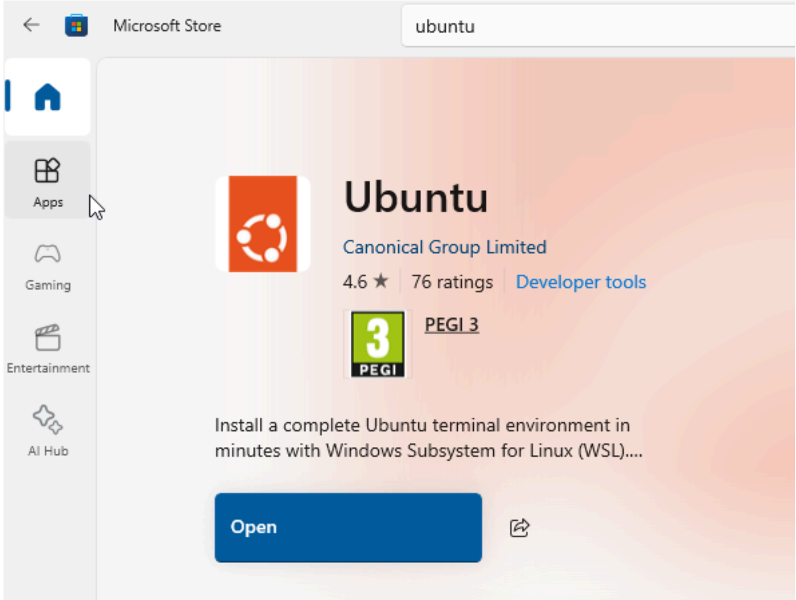
We are going to install Windows Subsystem for Linux, because building software on Windows directly is a bad experience :(



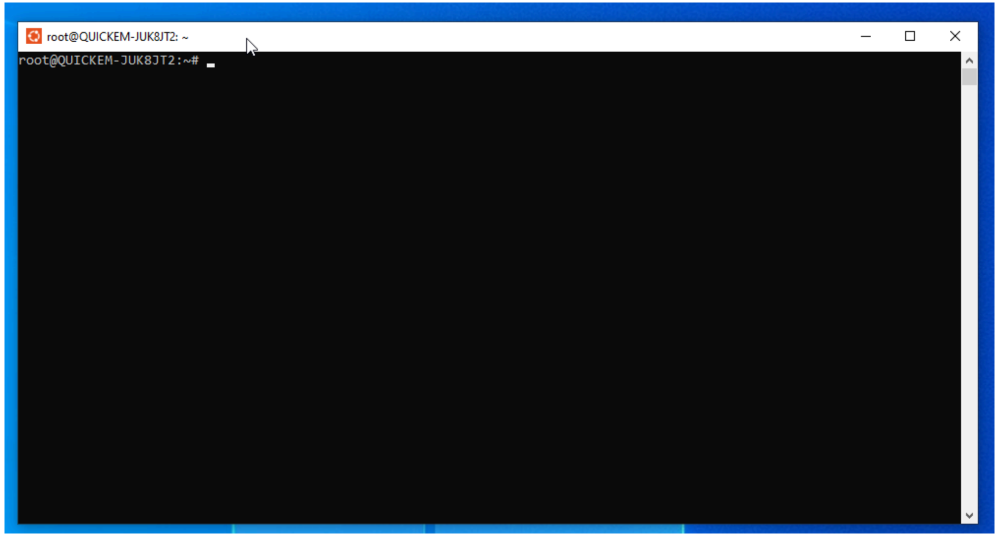


Check the box to enable Windows Subsystem for Linux, and restart your computer when prompted

5.1. Setup Windows



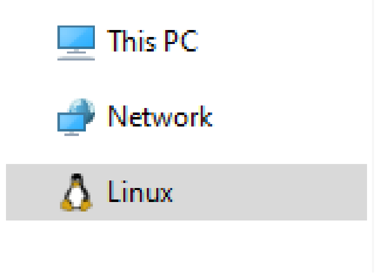
Open the Microsoft Store then search and install `Ubuntu`. Click open when it appears and wait for the installation to finish.



You will now have a linux terminal inside windows, so you can use linux commands and features.

5.1. Setup Windows

We can see the file system for this linux terminal inside file explorer by clicking on linux.



We can now use (debian) linux commands to install build tools from our Ubuntu terminal

- `$ sudo add-apt-repository ppa:team-gcc-arm-embedded/ppa`
- `$ sudo apt update`
- `$ sudo apt upgrade`
- `$ sudo apt install cmake gcc-arm-none-eabi build-essential`

5.2. Setup MacOS

- Install Homebrew:
  - https://brew.sh

Run the following commands in a terminal:

- `$ brew install cmake`
- `$ brew install --cask gcc-arm-embedded`

5.3. Setup Linux

Debian:

- `$ sudo add-apt-repository ppa:team-gcc-arm-embedded/ppa`
- `$ sudo apt update`
- `$ sudo apt upgrade`
- `$ sudo apt install cmake gcc-arm-none-eabi build-essential`

Arch:

- `$ sudo pacman -S git`
- `$ sudo pacman -S make`
- `$ sudo pacman -S cmake`
- `$ sudo pacman -S arm-none-eabi-gcc`
- `$ sudo pacman -S arm-none-eabi-newlib`

5.4. Setting up Picotool

First do

- `git clone https://github.com/raspberrypi/picotool`
- `cd picotool`

To avoid having to set a path variable for the pico-sdk you can edit `CMakeLists.txt` and add the following below first line (`cmake_minimum_required()`)

```cmake
set(PICO_SDK_FETCH_FROM_GIT 1)
include(pico_sdk_import.cmake)
```

5.4. Setting up Picotool

Then copy the file `pico_sdk_import.cmake` from sample repository into the picotool folder. Alternatively you can setup pico-sdk path variable.

Now run:

- `mkdir build`
- `cd build`
- `cmake ..`
- `make install`

Now you can use picotool to upload your code in the original repository.

5.4. Setting up Picotool

Note: For Windows users, you will need to go into the previously mentioned linux folder and then you will find your files inside the home folder.

5.5. PicoSDK Environment

- `git clone https://github.com/misskacie/microcontroller-workshop`
- `cd microcontroller-workshop`

We will be using this preconfigured environment I prepared for the rest of the workshop today.

Note: You can access my other repository https://github.com/misskacie/simple-pico-examples for fully setup examples at the end of today.

5.6. Building

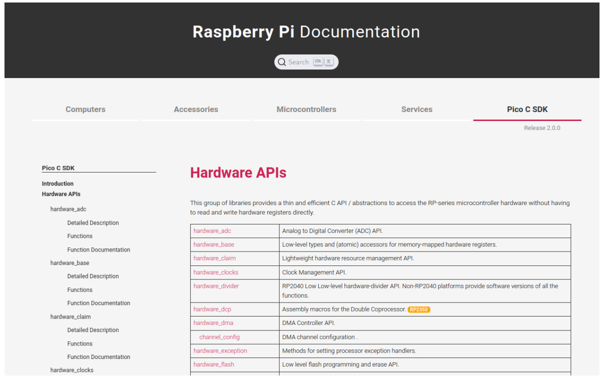
Run the following commands in a terminal within the project directory:

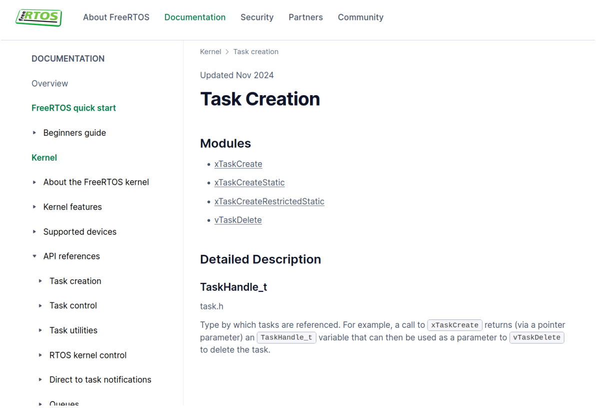
- `$ mkdir build`
- `$ cd build`
- `$ cmake ..`
- `$ make`

This creates a folder called build, then enters the folder and generates the cmake/make files and builds the project.

6. Practical Examples with Hardware

6.1. Documentation



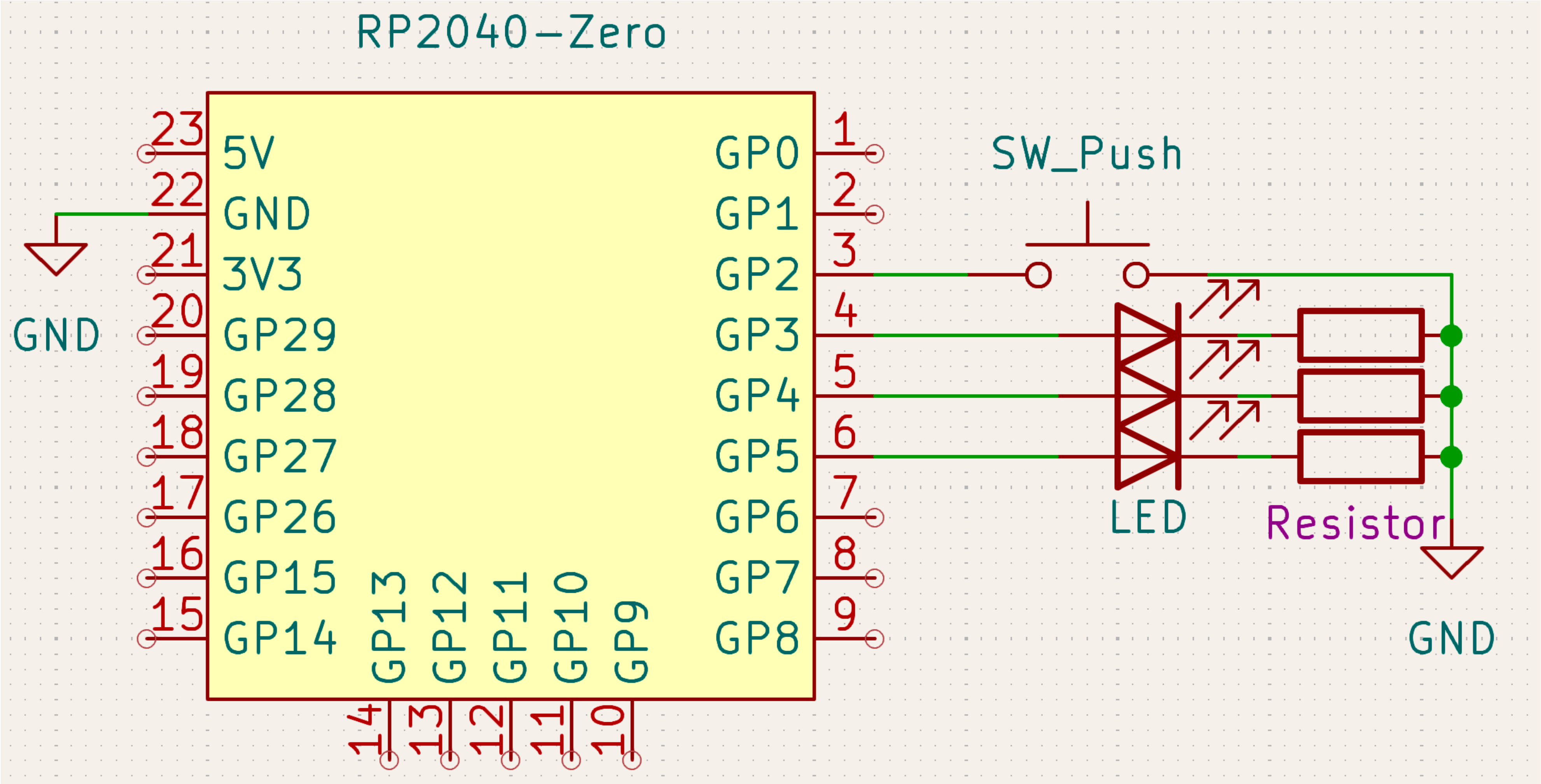


https://datasheets.raspberrypi.com/pico/raspberry-pi-pico-c-sdk.pdf

https://www.raspberrypi.com/documentation/pico-sdk/hardware.html

https://www.freertos.org/Documentation/00-Overview

6.2. Hardware Setup



6.3. Hello World Example

```c
#include <stdio.h>
#include "pico/stdlib.h"

int main() {
    stdio_init_all();

    while (true) {
        printf("Hello, world!\n");
        sleep_ms(1000);
    }
}
```

6.4. Building

Lets try building `helloworld.c`. Run the following commands in a terminal within the project directory:

- `$ mkdir build`
- `$ cd build`
- `$ cmake ..`
- `$ make`

6.4. Building

If we look in the build folder we should see the following files now.

- The most important file for us is `hello-world-example.uf2`. This is the file we are uploading to the microcontroller.

```
├── build/
│   ├── CMakeCache.txt
│   ├── CMakeFiles/
│   ├── cmake_install.cmake
│   ├── compile_commands.json
│   ├── _deps/
│   ├── generated/
│   ├── hello-world-example.bin*
│   ├── hello-world-example.dis
│   ├── hello-world-example.elf*
│   ├── hello-world-example.elf.map
│   ├── hello-world-example.uf2
│   ├── Makefile
│   ├── pico_flash_region.ld
│   ├── pico-sdk/
│   ├── pioasm/
│   └── pioasm-install/
```

6.5. Uploading the Code

There will now be a uf2 file within the build folder that can be flashed to the pico.

To flash the uf2 file to the pico you can hold down the boot button on the pico and then plug it in which will mount it like a storage device, then you copy the uf2 file onto the mounted storage device.

Alternatively you can use pico-tool from the Raspberry Pi Foundation which we setup earlier.

6.5. Uploading the Code

From inside the build folder you can run:

- `cmake..; make; sudo picotool load -f [filetoflash].uf2; sudo picotool reboot`

to automatically build and copy it over to the microcontroller, if it fails you will need to manually put it into bootsel mode as per previous slide.

6.6. GPIO Example

```c
#include <stdio.h>
#include "pico/stdlib.h"
#include "hardware/gpio.h"

#define LED_PIN 3

int main() {
    stdio_init_all();

    gpio_init(LED_PIN);
    gpio_set_dir(LED_PIN, GPIO_OUT);

    while(1) {
        gpio_put(LED_PIN, 0);
        sleep_ms(1000);
        gpio_put(LED_PIN, 1);
        sleep_ms(1000);
    }
}
```

6.7. CMake

So now that we have had a look at some examples, lets look at the `CMakeLists.txt` file I've set up for you so far.

This file generates the build instructions so that it can be compiled in the correct format without much effort when we run `cmake`.

6.7. CMake

CMake

```cmake
cmake_minimum_required(VERSION 3.27)

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

set(PICO_BOARD pico CACHE STRING "Board type")
set(PICO_PLATFORM rp2040)
set(PICO_SDK_FETCH_FROM_GIT 1)
set(FREERTOS_FETCH_FROM_GIT 1)
# Pull in Raspberry Pi Pico SDK (must be before project)
include(pico_sdk_import.cmake)

if (PICO_SDK_VERSION_STRING VERSION_LESS "2.0.0")
  message(FATAL_ERROR "Raspberry Pi Pico SDK version 2.0.0 (or later) required. Your version is ${PICO_SDK_VERSION_STRING}")
endif()

if (NOT DEFINED PICO_STDIO_USB_CONNECT_WAIT_TIMEOUT_MS)
    set(PICO_STDIO_USB_CONNECT_WAIT_TIMEOUT_MS 3000)
endif()

# Initialise the Raspberry Pi Pico SDK
pico_sdk_init()

project(simple-pico-examples C CXX ASM)
```

6.7. CMake

This first part is just alot of boiler plate code that you generally won't need to really touch.

- A couple of the main things to note is that in the case of these examples, I have it setup to automatically add the dependencies like the Pico-SDK and FreeRTOS using other CMake files such as `pico_sdk_import.cmake`, so we have these variables such as `PICO_SDK_FETCH_FROM_GIT` set to 1 for enabled.
- We also create a variable called `PICO_STDIO_USB_CONNECT_WAIT_TIMEOUT_MS`, without it we will have some weirdness with serial ports.

6.7. CMake

CMake

```cmake
add_executable(hello-world-example hello-world-example.c )

pico_set_program_name(hello-world-example "hello-world-example")
pico_set_program_version(hello-world-example "0.1")

# Modify the below lines to enable/disable output over UART/USB
pico_enable_stdio_uart(hello-world-example 0)
pico_enable_stdio_usb(hello-world-example 1)

# Add the standard include files to the build
target_include_directories(hello-world-example PRIVATE
        ${CMAKE_CURRENT_LIST_DIR}
        ${CMAKE_CURRENT_LIST_DIR}/.. # for our common lwipopts or any other standard includes, if required
)
# Add the required libraries
target_link_libraries(hello-world-example
        pico_stdlib
)

pico_add_extra_outputs(hello-world-example)

```

Now this next part is generating an output file for the microcontroller to run from `helloworld.c` file. These are called `uf2` files.

- We enable usb support since we want to read the serial monitor
- Notice that we need to link against `pico_stdlib` because we used the standard library, and we are adding an extra output (`uf2 file`).

6.7.1. Lets try add a new ouput for GPIO Example

First we can copy paste the CMake used for the hello world.

- What do we need to change?
- What libraries are we linking against?

Note: we can build a specific output such as `gpio-example` to avoid building every example again with:`$ make gpio-example`

6.7. CMake

In this case of the Pico-SDK we can find the name of the library to link against by looking at the folder names inside:

- `build/pico-sdk/src/rp2-common`
- `build/pico-sdk/src/rp2040`

Alternatively have a look at the documentation at:

- https://www.raspberrypi.com/documentation/pico-sdk/hardware.html

With minimal changes we can make the code build the gpio-example

CMake

```cmake
add_executable(gpio-example gpio-example.c )

pico_set_program_name(gpio-example "gpio-example")
pico_set_program_version(gpio-example "0.1")
# Modify the below lines to enable/disable output over UART/USB
pico_enable_stdio_uart(gpio-example 0)
pico_enable_stdio_usb(gpio-example 1)

# Add the standard include files to the build
target_include_directories(gpio-example PRIVATE
        ${CMAKE_CURRENT_LIST_DIR}
        ${CMAKE_CURRENT_LIST_DIR}/.. # for our common lwipopts or any other standard includes, if required
)

# Add the required libraries
target_link_libraries(gpio-example
        pico_stdlib
        hardware_gpio
)

pico_add_extra_outputs(gpio-example)
```

6.8. Interupts Example


```c
#include <stdio.h>
#include "pico/stdlib.h"
#include "hardware/gpio.h"
#include "hardware/clocks.h"
#include "hardware/timer.h"
#define BUTTON_PIN 2
#define LED_PIN 3
// Debounce control
int state = 0;
const int delayTime = 100000; // Delay for every push button may vary
absolute_time_t time, curr_time;
void gpio_callback(uint gpio, uint32_t events) {
    curr_time = get_absolute_time(); // All the timer stuff is for debouncing
    if (absolute_time_diff_us(time, curr_time) > delayTime) {
        time = get_absolute_time();
        state = !state;
        gpio_put(LED_PIN, state);
    }
}
```

6.8. Interupts Example

```c

int main() {
    stdio_init_all();
    time = get_absolute_time();

    gpio_init(BUTTON_PIN);
    gpio_pull_up(BUTTON_PIN);
    gpio_set_irq_enabled_with_callback(BUTTON_PIN, GPIO_IRQ_EDGE_FALL, true, &gpio_callback);

    gpio_init(LED_PIN);
    gpio_set_dir(LED_PIN, GPIO_OUT);
    gpio_put(LED_PIN, 0);

    while (1)
        printf("Hello GPIO IRQ\n");
}
```

Let's add a new CMake entry for this code!

6.9. Multi-Core Example

C

```c
#include <stdio.h>
#include "pico/stdlib.h"
#include "pico/multicore.h"
#define LED_PINA 3
#define LED_PINB 4

void core1_entry() {
    while(1) {
        printf("Hello Core1\n");
        gpio_put(LED_PINB, 0);
        sleep_ms(1000);
        gpio_put(LED_PINB, 1);
        sleep_ms(1000);
    }
}
```

6.9. Multi-Core Example

```
int main() {
    stdio_init_all();
    printf("Hello, multicore!\n");
    multicore_launch_core1(core1_entry);

    gpio_init(LED_PINA);
    gpio_init(LED_PINB);
    gpio_set_dir(LED_PINA, GPIO_OUT);
    gpio_set_dir(LED_PINB, GPIO_OUT);

    while(1) {
        printf("Hello Core0\n");
        gpio_put(LED_PINA, 0);
        sleep_ms(1000);
        gpio_put(LED_PINA, 1);
        sleep_ms(1000);
    }
}
```

Lets add another CMake entry the same as before!

6.10. DMA Example

C

```c
#include <stdio.h>
#include "pico/stdlib.h"
#include "hardware/dma.h"

// Data will be copied from src to dst
const char src[] = "Hello, world! (from DMA)";
char dst[count_of(src)];

int main() {
    stdio_init_all();

    int chan = dma_claim_unused_channel(true);

    // 8 bit transfers. Both read and write address increment after each
    // transfer (each pointing to a location in src or dst respectively).
    // No DREQ is selected, so the DMA transfers as fast as it can.

    dma_channel_config c = dma_channel_get_default_config(chan);
    channel_config_set_transfer_data_size(&c, DMA_SIZE_8);
    channel_config_set_read_increment(&c, true);
    channel_config_set_write_increment(&c, true);

```

6.10. DMA Example

```
    dma_channel_configure(
        chan,          // Channel to be configured
        &c,            // The configuration we just created
        dst,           // The initial write address
        src,           // The initial read address
        count_of(src), // Number of transfers; in this case each is 1 byte.
        true           // Start immediately.
    );
    // We could choose to go and do something else whilst the DMA is doing its
    // thing. In this case the processor has nothing else to do, so we just
    // wait for the DMA to finish.
    dma_channel_wait_for_finish_blocking(chan);

    // The DMA has now copied our text from the transmit buffer (src) to the
    // receive buffer (dst), so we can print it out from there.
    puts(dst);
}

```

Once again lets add a compile target to the CMake, and test it out!

6.11. Watchdog Example

```c
#include <stdio.h>
#include "pico/stdlib.h"
#include "hardware/watchdog.h"

int main() {
    stdio_init_all();

    if (watchdog_caused_reboot()) {
        printf("Rebooted by Watchdog!\n");
        return 0;
    } else {
        printf("Clean boot\n");
    }

    // Enable the watchdog, requiring the watchdog to be updated every 100ms or the chip will reboot
    // second arg is pause on debug which means the watchdog will pause when stepping through code
    watchdog_enable(100, 1);

    for (uint i = 0; i < 5; i++) {
        printf("Updating watchdog %d\n", i);
```

6.11. Watchdog Example

```
            watchdog_update();
        }

        // Wait in an infinite loop and don't update the watchdog so it reboots us
        printf("Waiting to be rebooted by watchdog\n");
        while(1);
    }
```

Once again lets add a compile target to the CMake, and test it out!

6.12. FreeRTOS Example

```c
#include <FreeRTOS.h>
#include <task.h>
#include <stdio.h>
#include "pico/stdlib.h"

#define LED_PINA 3
#define LED_PINB 4
#define LED_PINC 5
struct led_task_arg {
    int gpio;
    int delay;
};

void led_task(void *p) {
    struct led_task_arg *a = (struct led_task_arg *)p;

    gpio_init(a->gpio);
    gpio_set_dir(a->gpio, GPIO_OUT);
    while (true) {
        gpio_put(a->gpio, 0);
        vTaskDelay(pdMS_TO_TICKS(a->delay));
        gpio_put(a->gpio, 1);
        vTaskDelay(pdMS_TO_TICKS(a->delay));
    }
}
```

6.12. FreeRTOS Example

```

int main()
{
    stdio_init_all();

    printf("Start LED blink\n");

    struct led_task_arg arg1 = {LED_PINA, 1000};
    xTaskCreate(led_task, "LED_Task 1", 256, &arg1, 1, NULL);

    struct led_task_arg arg2 = {LED_PINB, 2000};
    xTaskCreate(led_task, "LED_Task 2", 256, &arg2, 1, NULL);

    struct led_task_arg arg3 = {LED_PINC, 4000};
    xTaskCreate(led_task, "LED_Task 3", 256, &arg3, 1, NULL);

    vTaskStartScheduler();

    while (true);
}
```

CMake linking in this case is a bit more complicated, we need to link against FreeRTOS. Like the Pico-SDK, importing is mostly automatic.

6.12. FreeRTOS Example

CMake

```cmake
add_library(freertos_config INTERFACE)
target_include_directories(freertos_config INTERFACE
        "${CMAKE_CURRENT_LIST_DIR}/FreeRTOS"
)
include(FreeRTOS/FreeRTOS_import.cmake)
include(FreeRTOS/FreeRTOS_Kernel_import.cmake)
add_executable(freertos-example freertos-example.c FreeRTOS/FreeRTOSPortAux.c)
pico_set_program_name(freertos-example "freertos-example")
pico_set_program_version(freertos-example "0.1")

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

# Add the standard include files to the build
target_include_directories(freertos-example PRIVATE
        ${CMAKE_CURRENT_LIST_DIR}
        ${CMAKE_CURRENT_LIST_DIR}/.. # for our common lwipopts or any other standard includes, if required
)

# Add the required libraries
target_link_libraries(freertos-example
        pico_stdlib FreeRTOS-Kernel FreeRTOS-Kernel-Heap4 freertos_config
)
pico_add_extra_outputs(freertos-example)
```

6.13. More Examples

You can have look at this repository from the Raspberry Pi Foundation:

- https://github.com/raspberrypi/pico-examples

Do note that the folder structure in that repository is a bit messy with each folder having its own CMakeLists.txt and file to build!