



Institut Supérieur de l'Aéronautique et de l'Espace ISAE-SUPAERO
Supaéro Space Section

---

Sparrow

# Embedded Systems Programming

Florian Topeza
November 28th, 2024

---

# Contents

# List of Figures

# List of Tables

# Introduction

This document is the course material for *Embedded systems programming* taught at ISAE-SUPAERO as part of the Sparrow 2024 training project by the Supaero Space Section.

This document covers all the lecture content, with some additional insights. These insights, highlighted by a bold title and a vertical line on the left, provide some avenues for reflection on topics covered by the course but are not necessary for the Sparrow project.

Additional resources for the course, including MicroPython code, are available in this GitHub repository.

For any comments, suggestions, or corrections regarding this document, please contact me so I can update and correct this document.

# Part I

# Raspberry Pi Pico Microcontroller

## I.1 Introduction

To fly, a Sparrow rocket must include electronic components: a servo motor to open and close the parachute hatch, a buzzer whose sound indicates the rocket's status (ready for launch, in flight...), and sensors (IMU and barometer) to collect flight data. These various elements need to be controlled in order to function.

What should we use to control the rocket's electronics? We need something compact to fit inside the rocket and energy-efficient. A computer is obviously out of the question. A micro-computer (like those from Raspberry) is more compact, but it consumes too much energy! Indeed, even without an active program to control the rocket, the micro-computer's operating system (OS) consumes energy. That's why we'll be using a microcontroller.

A microcontroller, unlike a micro-computer, does not have an operating system and can be programmed by simply uploading only the necessary programs. Often, a microcontroller can be programmed in various ways, with varying levels of complexity. In the case of Sparrow, we'll use an integrated development environment (IDE) that allows us to write and upload code to our board simply.

We still need to choose the model of microcontroller to use. We need a microcontroller with enough memory to store the programs we'll upload to it (for Sparrow, this represents a few kilobytes), but especially one that can store the flight data recorded by the sensors. We also need a processor capable of performing the tasks we ask of it quickly and simultaneously. Finally, it should be a user-friendly microcontroller with good documentation available. This last point might seem trivial, but it is crucial: in any project, it's better to use well-tested equipment with resources to guide us rather than unknown equipment with uncertain operation (in the space sector, this is even formalized by the concept of Technology Readiness Level, or TRL).

With all these elements in mind, the microcontroller we'll be using is the Raspberry Pi Pico.

## I.2 Presentation of the Microcontroller


Figure 1: Raspberry Pi Pico

The Raspberry Pi Pico uses an RP2040 processor developed by Raspberry Pi. This is the black square visible at the center of the board. To better understand the capabilities of the Raspberry Pi Pico and its RP2040 microcontroller, it is useful to compare it with

a mid-range computer and an Arduino Nano, a microcontroller from the Arduino family equivalent in size to the Pico. Here is a comparison table that shows the major differences between these three systems.

| **Features** | **Raspberry Pi Pico (RP2040)** | **Arduino Nano** | **Mid-Range Computer** |
|---|---|---|---|
| Processor (CPU) | 2 cores ARM Cortex-M0+ at 133 MHz | 1 core ATmega328P at 16 MHz | x86_64 processor (e.g., Intel Core i5) at 2.5-4 GHz |
| Architecture | ARM Cortex-M0+ 32 bits | AVR 8 bits | x86_64 64 bits |
| RAM | 264 KB SRAM | 2 KB SRAM | 8 GB - 16 GB DDR4 |
| Flash Memory | 2 MB Flash | 32 KB Flash | 256 GB - 1 TB storage (SSD/HDD) |
| Bus Speed | Up to 133 MHz | 16 MHz (MCU frequency) | Several GHz |
| Input/Output (I/O) | 26 GPIO, 2 × UART, 2 × SPI, 2 × I2C, 3 × ADC | 14 GPIO (6 PWM), 1 × UART, 1 × SPI, 1 × I2C, 8 ADC | USB 3.0, Ethernet, HDMI, etc. |
| Power Consumption | 1.8 V - 3.3 V | 5 V (via USB or Vin) | 100 W - 400 W (depending on usage) |
| Operating Systems | Bare metal, MicroPython, FreeRTOS | Bare metal (Arduino IDE) | Windows, macOS, Linux |
| Cost | 4 USD | 20 USD | 500 - 1500 USD |

Table 1: Comparison of the Raspberry Pi Pico's performance with the Arduino Nano and a mid-range computer

The Raspberry Pi Pico offers much better performance and more versatility than an Arduino Nano: more pins to connect components, its 2-core processor is faster and more powerful than the Nano's processor, its 2MB Flash memory allows longer scripts to be uploaded and data to be written to a file on the board, which is impossible on the Nano, and the Pico offers far more programming options, being programmable in MicroPython, C/C++. And all this while being cheaper than the Nano!

**What's a core in a processor?**

> Let's use an analogy. Imagine your processor is a kitchen, and each core in your processor is a chef. If your processor has only one core, i.e., there's only one chef in the kitchen, and you have multiple dishes to prepare, your chef will have to switch between dishes; they can't handle all the dishes simultaneously. However, if you have multiple chefs, you can prepare several dishes at the same time without risking burning one while tending to another!
> This is exactly what happens with processor cores. If there's only one core, it will switch between tasks, advancing each one in small steps—this is called multi-threading. The risk here is that a resource-intensive task might monopolize the core, preventing other tasks from progressing. With multiple cores, a resource-intensive task can be handled by a specific core without hindering other tasks, which are carried out by the other cores.

**For the curious...**

> I recommend this YouTube video (and this channel in general) if you enjoy electronics and want to deepen your understanding of how the microcontroller works. You can also find the RP2040 documentation here.

## I.3 Operation of the Raspberry Pi Pico Pins

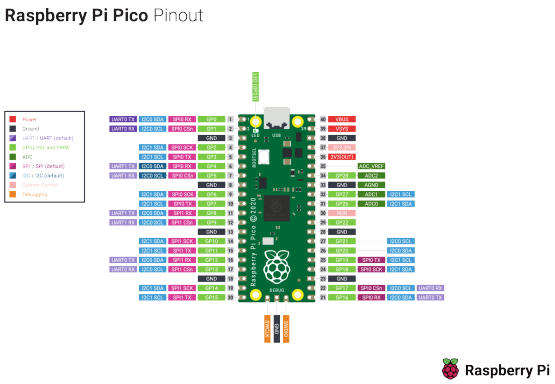



Figure 2: Raspberry Pi Pico Pinout

The Raspberry Pi Pico is equipped with a set of pins offering a wide range of functionalities for electronic project development. These pins allow connection to various sensors, modules, and other peripherals to interact with the RP2040 microcontroller. Here is an explanation of the different functions of the Raspberry Pi Pico pins.

**Programming of UART, SPI, I2C, and PWM protocols is explained in the Classes section of the MicroPython Programming part of this course.**

### I.3.1 GPIO (General-Purpose Input/Output)

The GPIO pins can be configured as digital inputs or outputs. This allows reading the state of a button, turning on an LED, or controlling other electronic components. The Raspberry Pi Pico has 26 GPIOs numbered from 0 to 25. Each GPIO can be configured in input, output, or in some cases, analog input/output mode.

### I.3.2 ADC (Analog-to-Digital Converter)

The ADC pins allow reading analog signals, i.e., variable values such as the voltage from an analog sensor. GPIO 26, 27, and 28 are the ADC pins, capable of reading analog signals with 12-bit resolution. Reading from analog sensors like potentiometers, temperature sensors, or light sensors.

### I.3.3 UART (Universal Asynchronous Receiver/Transmitter)

UART pins are used for serial communication with other microcontrollers, computers, or serial peripherals like GPS or Bluetooth modules.

There is no shared clock signal between devices, meaning synchronization is managed by the baud rate, which must be configured identically on both sides.

The Pico's TX pin should be connected to the RX pin of the device, and vice versa. For example, if connecting a GPS module to the Pico, the TX of the GPS goes to the RX of the Pico and the RX of the GPS goes to the TX of the Pico. On some devices, however, pins need to be connected directly without crossing, with TX to TX and RX to RX, though this is quite rare.

The Pico operates at 3.3V, so ensure connected devices are also 3.3V compatible or use level shifters if needed, as some peripherals use 5V UART, for instance.

The Pico has two UART controllers. GPIO 0 (TX) and 1 (RX) for UART0, and GPIO 4 (TX) and 5 (RX) for UART1.

### I.3.4 SPI (Serial Peripheral Interface)

The SPI protocol is used for fast communication with devices like displays, flash memory, or sensors. This synchronous serial communication protocol enables fast communication between a master (like the Pico) and one or more slave devices. SPI uses four wires: TX/MOSI, RX/MISO, SCK/CLK, and CS/SS.

MOSI (Master Out Slave In) is the line where the master sends data, MISO (Master In Slave Out) is the line where the master receives data, SCK (Serial Clock) is the clock line that synchronizes data transfer, and CS (Chip Select) is the line that selects the active slave.

The master's MOSI pin should be connected to each slave's MOSI pin, and the master's MISO pin should be connected to each slave's MISO pin. The SCK pin must be shared among all devices, and each slave must have its own SS pin connected to the master.

The CS (Chip Select) pin should be used to select the slave with which the master wants to communicate. Each slave should be configured to be selected by setting its CS pin to LOW.

The Pico has two SPI controllers. The available pins for SPI0 are GPIO 16 (TX/-MOSI), 17 (RX/MISO), 18 (SCK/CLK), 19 (CS/SS), while for SPI1, the available pins are GPIO 12 (TX/MOSI), 13 (RX/MISO), 14 (SCK/CLK), 15 (CS/SS).

### I.3.5 I2C (Inter-Integrated Circuit)

I2C is a communication protocol that enables connecting multiple devices with minimal wiring. It is ideal for sensors and modules requiring bidirectional communication. This protocol is commonly used for communication with RTC modules, temperature sensors, or LCD screens.

SDA (Serial Data Line) is the line where data is transmitted. SCL (Serial Clock Line) is the clock line that synchronizes data transfers.

The SDA and SCL lines should be connected to the device's SDA and SCL lines, respectively. For example, when connecting an I2C LCD screen, the Pico's SDA goes to the LCD's SDA, and the Pico's SCL goes to the LCD's SCL.

The SDA and SCL lines need pull-up resistors to function properly. The Pico includes internal pull-up resistors, but for longer bus lengths or high speeds, external resistors (typically 4.7kΩ) are recommended.

The Pico has two I2C controllers. Available pins for I2C0 are GPIO 4 (SDA), 5 (SCL), and for I2C1, GPIO 6 (SDA), 7 (SCL).

### I.3.6 PWM (Pulse Width Modulation)

PWM (Pulse Width Modulation) is a technique used to simulate an analog voltage using a digital signal. PWM works by generating a square wave with a fixed frequency but with variable pulse width (i.e., the duration when the signal is at a high level).

The proportion of time the signal is high relative to the total signal period is called the "duty cycle." For example, a 50% duty cycle means the signal is high 50% of the time and low the remaining 50%. By adjusting this ratio, PWM can control the intensity of a current, the speed of a motor, the brightness of an LED, etc.

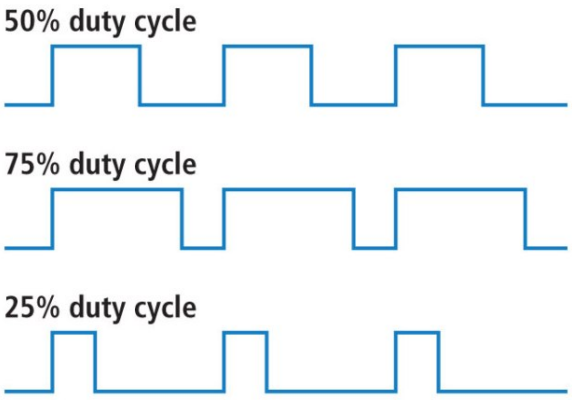
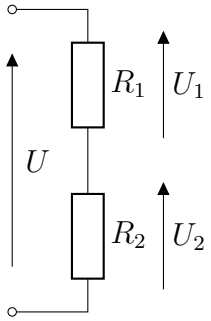



Figure 3: PWM signals with different duty cycles

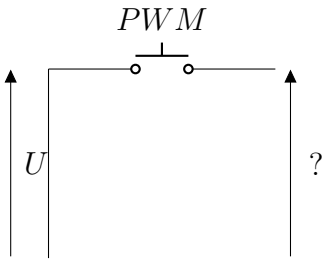
All GPIOs can be used for PWM (up to 16 PWM channels in total). PWM enables controlling motor speeds, servo positions, or brightness.

**Voltage Regulation Application**

Suppose we have a power supply providing a voltage $U$ and we want to divide this voltage by two to power a system. A simple first idea would be to use a voltage divider.

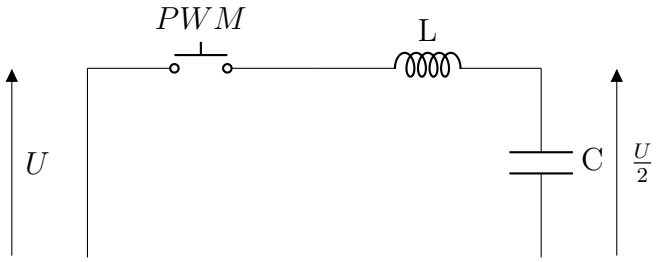



Since $U_2 = \dfrac{R_2}{R_1 + R_2} U$, choosing $R_1 = R_2$ gives $U_2 = \dfrac{U}{2}$.
However, such a device is inefficient due to Joule effect losses. A much more efficient, but subtle strategy is to use a PWM-controlled switch with a 50% duty cycle.




The output voltage is then a square wave signal, whose average value (signal integral over a period, divided by the period) is half the input voltage. We aim to process this square wave signal to retain only its average value, also known as the DC component. All harmonics of the signal are filtered out using a low-pass filter with a cutoff frequency at least 10 times lower than the fundamental frequency of the square wave, which is $f_0 = \frac{1}{T}$, where $T$ is the PWM period.

We choose an LC filter rather than an RC filter to avoid Joule effect losses. In signal processing, an RC filter would instead be preferred to avoid magnetic phenomena caused by inductance. To design L and C, we need $f_c = \frac{1}{2\pi\sqrt{LC}} \leq \frac{f_0}{10}$.




We then obtain an output voltage equal to half of the input voltage, with significantly higher efficiency than a resistive voltage divider. This principle is the basis for switch-mode voltage regulators, whose efficiency can reach up to 95

### I.3.7 Power

**VBUS (USB Power Input)** is connected to the VSYS pin through a Schottky diode, allowing current to pass from the VBUS pin to the VSYS pin while blocking current in the opposite direction. When the board is powered via its micro USB connector, the VBUS pin powers the board and provides a 5V output that can be used to power other components.

**In summary, VBUS supplies 5V only when the board is connected to USB, and this voltage can be used to power other external components. Be cautious, as connecting VBUS to ground while the board is powered via USB will result in a short circuit, and the board will not function!**

**VSYS (System Power In)** is the primary power pin for the Raspberry Pi Pico. When the board is powered through its micro USB port, the current from VBUS flows to VSYS via the Schottky diode, powering the board. When powered through another source, such as a battery, the power should be connected to the VSYS pin, which directly powers the board.

**In summary, VSYS is the main power pin for the Raspberry Pi Pico, and it should receive an external voltage between 1.8V and 5.5V or be powered through VBUS when USB is connected. This pin does not supply current.**

**3V3(OUT) (3.3V Output)** is a regulated 3.3V output used to power external components.

Lastly, the **GND** (Ground) pins are essential to complete electrical circuits and provide a common voltage reference. The ground pins of the circuit components should be connected to the ground pins of the Pico.

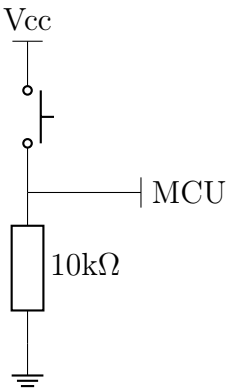
### I.3.8 System Control

**3V3_EN (Enable 3.3V Regulator)** is an input that controls the Pico's 3.3V regulator. When pulled to ground (GND), it disables the 3.3V regulator, cutting power to both the RP2040 and any devices connected to the 3V3 pin. By default, this pin is internally connected to VSYS through a pull-up resistor, keeping the regulator active. This pin can be used to disable the 3.3V power supply in scenarios requiring reduced power consumption, such as in low-power applications or when the board should enter a deep sleep mode.

**In summary, if this pin is connected to ground, the board stops functioning.**

**Pull-up and Pull-down Resistors**

> Pull-up and pull-down resistors are resistors placed to prevent pins from floating, meaning the pin voltage randomly varies when nothing is connected. This can happen, for example, when a pin is connected to a push button. When the button is pressed, the pin voltage equals the button's supply voltage. However, when released, the pin voltage may not necessarily return to zero, leading to inaccurate signal readings! A pull-down resistor connects the pin to ground when no signal is received, while a pull-up resistor keeps the pin at a high voltage when disconnected.

> Here is an example of a pull-down resistor circuit. When the switch is closed, current flows along the path of least resistance, and the microcontroller pin receives the Vcc voltage. When the switch is open, the pin is grounded.
>
> 
> 
>
> The Raspberry Pi Pico pins have built-in pull-up or pull-down resistors that can be activated in code, as described here.

**RUN (Reset)** is an input that controls the board's power. When pulled to ground (GND), it disables the 3.3V regulator, cutting off power to the board and putting the RP2040 in "off" mode. Releasing this pin (leaving it floating or connected to VSYS via a resistor) restarts the regulator, powering the board again. This pin is used to perform a hardware reset or put the board into standby (minimal power consumption) by disabling power.

**In summary, if this pin is grounded, the board stops functioning.**

### I.3.9 SWD (Serial Wire Debug)

The SWD pins are used for real-time code debugging and advanced programming on the Raspberry Pi Pico. They are located at the SWDIO and SWCLK pins.

### I.3.10 Internal LED

The Raspberry Pi Pico includes an onboard LED connected to GPIO 25. This LED can be used as a status indicator or for code tests. It is helpful for checking the microcontroller's operation or running quick code tests.

**For More Information...**

> Many resources are available to explore the Raspberry Pi Pico's capabilities and other microcontrollers. This tutorial revisits this course section with additional technical details for those of you interested in a deeper understanding.

# Part II

# Programming in MicroPython

## II.1 Thonny

To upload code to the Raspberry Pi Pico, we will use the Thonny IDE, which can be downloaded here. To connect the Raspberry Pi Pico to the computer, you'll need a USB-A to micro-USB cable capable of data transfer. Be cautious, as some cables only allow charging and cannot transfer data.

Once Thonny is open and the cable is connected to the computer, hold down the BOOTSEL button on the Pico while plugging the board into the cable. This puts the board into USB mass storage mode, and a file explorer window should open. This window can be useful for debugging the board, but it is not needed here.

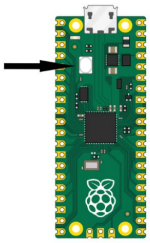


Figure 4: Location of the BOOTSEL button on the board

In the bottom-right corner of the Thonny window, you can see the version of Python currently in use. Click on the version and select MicroPython (Raspberry Pi Pico). If this option does not appear, verify the Pico's connection.

Now look at the console at the bottom of the Thonny editor; you should see something similar to the following screenshot.


Figure 5: Thonny Shell

Thonny is now ready to communicate with the board. To easily navigate between files on your computer and the Pico, ensure that the **Files** option is checked in the **View** menu at the top of Thonny.

To upload a Python file from your computer to the board, open the file in Thonny, go to **File** and select **Save As**, then save the file to the Pico. If the file is visible in the file tree on the left side of the Thonny editor, right-click on the file and select **Upload to** to transfer it to the board.

# II.2 Libraries

## II.2.1 General Reminders

A library in Python is a set of modules, i.e., Python files containing functions, classes, and variables that allow specific tasks to be accomplished. Libraries are used to avoid reinventing the wheel by providing reusable code for common functionalities, such as file manipulation, time management, hardware interaction, and more.

To use a library in Python, you must import it into your script. Importing a library is done with the keyword import. Here are some basic examples.

```
import math  # Imports the entire math library
import os    # Imports the entire os library
```

Once imported, you can access the library's functions and classes using the syntax `library_name.function_name()`. For example:

```
import math

result = math.sqrt(16)  # Uses the sqrt function from the
# math library to calculate the square root of 16

print(result)  # Displays 4.0
```

If you only want to import a specific part of a library, you can specify a particular module or function:

```
from math import sqrt  # Only imports the sqrt function
                       # from the math module

result = sqrt(25)
print(result)  # Displays 5.0
```

You can also rename a library or function during import:

```
import math as m  # Renames math to m

result = m.sqrt(9)
print(result)  # Displays 3.0
```

## II.2.2 Important Libraries Pre-installed on a Raspberry Pi Pico with MicroPython

The Raspberry Pi Pico, when used with MicroPython, is equipped with several essential libraries that allow interaction with hardware and management of basic tasks. Here are

some of the most important ones:

`machine`

The `machine` library is essential for interacting with the hardware on the Raspberry Pi Pico. It allows control of GPIO (General Purpose Input/Output), interfaces like I2C, SPI, UART, timers, PWM, etc.

```
from machine import Pin

led = Pin(25, Pin.OUT)  # Creates a Pin object to control
                        # the built-in LED

led.value(1)  # Turns on the LED
```

`time`

The `time` library provides functions for managing time, such as delays, elapsed time, etc. This library is often used to create pauses or to time events.

```
import time

time.sleep(2)  # Pause for 2 seconds
print("2 seconds have passed")
```

`utime`

`utime` is a MicroPython-specific version of the `time` library, offering similar functionality but optimized for microcontrollers.

```
import utime

start = utime.ticks_ms()  # Get the current time in
                          # milliseconds
utime.sleep_ms(100)       # Pause for 100 milliseconds
end = utime.ticks_ms()

print("Elapsed time: ", utime.ticks_diff(end, start), "ms")
```

`uos`

The `uos` library (similar to `os` in standard Python) is used to interact with the file system, access files and directories, etc.

```
import uos

uos.listdir()  # Lists files and directories in the
               # current directory
```

# II.3 Classes

## II.3.1 General Reminders

A **class** in Python is a template that allows creating objects. A class groups data in the form of attributes (variables) and behaviors in the form of methods (functions) associated with this data. Classes allow defining custom object types by grouping both the data and the functionalities that operate on this data.

To create a class in Python, the `class` keyword is used, followed by the name of the class and a colon (`:`). Inside the class block, we define methods, the first of which is usually `__init__`. This method is a constructor that initializes the objects created from the class. Here is a simple example:

```
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

    def display_details(self):
        print(f"Car: {self.make} {self.model},
        Year: {self.year}")
```

- **Class Declaration**: The `Car` class is created with three attributes: `make`, `model`, and `year`.
- **`__init__` Method**:
  - This special method is automatically called when a new object is created.
  - It initializes the object's attributes with the provided values.
  - The `self` parameter refers to the current instance of the class, i.e., the object itself.
- **`display_details` Method**:
  - This is a regular method of the class.
  - It displays the details of the car.
  - `self` is used to access the attributes of the object.

To use a class, you must first create an object (an instance) of that class. You can then access the attributes and call the object's methods. Example of usage:

```
# Creating an instance of the Car class
my_car = Car("Toyota", "Corolla", 2020)

# Calling the display_details method to display the
```

```
# car's information
my_car.display_details()
```

- **Creating the Object**: `my_car = Car("Toyota", "Corolla", 2020)`
  - Here, an object `my_car` is created from the `Car` class.
  - The values `"Toyota"`, `"Corolla"`, and `2020` are passed to the `__init__` constructor to initialize the object.
- **Calling a Method**: `my_car.display_details()`
  - This line calls the `display_details` method of the `my_car` object, which displays the car's information.

### II.3.2 Important Classes on a Raspberry Pi Pico

On a Raspberry Pi Pico, several important classes are available, especially when using MicroPython. These classes are essential for interacting with the board's hardware, controlling peripherals, and managing input/output. Here are the main classes to know.

#### Pin

The `Pin` class is used to control the GPIO (General Purpose Input/Output) pins of the Raspberry Pi Pico. You can configure each pin as a digital input or output and read or write values on these pins.

```
from machine import Pin

# Configuring a pin as output
led = Pin(25, Pin.OUT)
led.value(1)  # Turns on the LED connected to pin 25

# Configuring a pin as input
button = Pin(14, Pin.IN, Pin.PULL_DOWN)
button_state = button.value()  # Reads the state of the button (0 or 1)
```

- `Pin.OUT`: Configures the pin as output.
- `Pin.IN`: Configures the pin as input.
- `Pin.PULL_DOWN` / `Pin.PULL_UP`: Enables the internal pull-down or pull-up resistor on the pin. In the case of a pin configured as input, for example, to read the value of a push button (LOW if the button is released and HIGH if pressed), enabling the pull-down resistor prevents the pin from being left floating when the button is released.

### ADC

The `ADC` (Analog-to-Digital Converter) class is used to read analog values on certain pins of the Raspberry Pi Pico. This allows reading analog sensors like potentiometers or temperature sensors.

```
from machine import ADC

# Creating an ADC object on pin 26 (GP26)
potentiometer = ADC(26)

# Reading the analog value (between 0 and 65535)
value = potentiometer.read_u16()
print("Read value:", value)
```

- `ADC.read_u16()`: Returns a value between 0 and 65535 corresponding to the read voltage.

### PWM

The `PWM` (Pulse Width Modulation) class is used to generate PWM signals on the GPIO pins. PWM is often used to control the brightness of LEDs, motor speed, etc.

```
from machine import Pin, PWM

# Configuring pin 15 as PWM
led_pwm = PWM(Pin(15))

# Setting the PWM frequency
led_pwm.freq(1000)  # 1 kHz

# Setting the duty cycle between 0 (off) and 65535 (100% on)
led_pwm.duty_u16(32768)  # 50% brightness
```

- `PWM.freq()`: Sets the frequency of the PWM signal.
- `PWM.duty_u16()`: Sets the duty cycle using a value between 0 and 65535.

### I2C

The `I2C` class allows communication with devices using the I2C (Inter-Integrated Circuit) bus, a common serial communication protocol for sensors and other modules.

```
from machine import Pin, I2C

# Configuring the I2C on GPIO8 (SDA) and GPIO9 (SCL) pins
```

```
i2c = I2C(0, scl=Pin(9), sda=Pin(8), freq=400000)

# Scanning for connected I2C devices
devices = i2c.scan()
print("I2C devices found:", devices)

# Reading and writing data to an I2C device
# Example: read 2 bytes from address 0x3C
data = i2c.readfrom(0x3C, 2)
```

- `I2C.scan()`: Scans the I2C bus and returns a list of connected device addresses.
- `I2C.readfrom()`: Reads data from an I2C device.

### SPI

The `SPI` class is used to communicate with devices using the SPI (Serial Peripheral Interface) bus, another serial communication protocol used for displays, sensors, memory, etc.

```
from machine import Pin, SPI

# SPI configuration on GPIO10 (SCK), GPIO11
# (MOSI), GPIO12 (MISO)
spi = SPI(0, baudrate=1000000, polarity=0, phase=0,
          sck=Pin(10), mosi=Pin(11), miso=Pin(12))

# Writing and reading SPI data
# Example: write [0x01, 0x02] and read 2 bytes
spi.write([0x01, 0x02])
data = spi.read(2)
```

- In the initialization of SPI, the `baudrate` defines the data rate between the microcontroller and the device. The baudrate during initialization must match the baudrate used by the device.
- `SPI.write()` : Sends data over the SPI bus.
- `SPI.read()` : Reads data from the SPI bus.

### UART

The `UART` class is used for serial communication via UART pins. It is often used to communicate with serial modules, such as GPS, modems, or Bluetooth modules.

```
from machine import UART

# UART configuration on UART port 0, with pins
# GPIO0 (TX) and GPIO1 (RX)
uart = UART(0, baudrate=9600, tx=Pin(0), rx=Pin(1))

# Sending data
uart.write('Hello, UART!')

# Reading data
data = uart.read(10)  # Reads 10 bytes of data
```

- As with the SPI class, the chosen baudrate must be the one used by the device connected to the microcontroller.
- `UART.write()` : Sends data over the UART bus.
- `UART.read()` : Reads the data received by the UART bus.

**Timer**

The `Timer` class allows creating hardware timers, often used to execute functions at regular intervals without blocking the main program.

```
from machine import Timer

# Creating a timer that executes a function every
# 2 seconds
def clignotement(t):
    print("Timer triggered!")

timer = Timer()
timer.init(period=2000, mode=Timer.PERIODIC,
    callback=clignotement)
```

- `Timer.PERIODIC` : Periodic mode where the function is called at regular intervals.
- `Timer.ONE_SHOT` : Mode where the function is called only once after the specified delay.

## II.4 Writing Files in MicroPython on Raspberry Pi Pico

In MicroPython, it is possible to write data to a file on the internal memory of the Raspberry Pi Pico. This operation is useful for storing data persistently, such as logs, configurations, or sensor results.

To write data to a file in MicroPython, we use the `open()` function to open (or create) a file. Then, we use the `write()` method to write data into the file. Finally, it is important to close the file with `close()` to ensure all data is properly saved.

The `file.flush()` command allows you to clear the write buffer of a file and force immediate writing of data to disk. Normally, data written to a file may be temporarily stored in a buffer before being saved to disk. Using `flush()` is particularly useful to ensure that data is saved immediately, for example, if there is a risk of power failure or sudden program termination. This minimizes the risk of data loss.

```
# Open (or create) a file named "data.txt" in write mode
with open("data.txt", "w") as file:
    # Write data to the file
    file.write("Hello, Raspberry Pi Pico!\n")
    file.write("This is an example of writing to a file.\n")

# The file is automatically closed at the end of the with block
```

If you want to add data to an existing file without overwriting it, use the `'a'` mode:

```
# Open the file in append mode
with open("data.txt", "a") as file:
    # Add data to the file
    file.write("Adding new data.\n")
```

- `'w'` : Opens a file in write mode. If the file exists, its contents will be erased before the new write begins.
- `'a'` : Opens a file in append mode. New data will be added at the end of the file without deleting the existing content.
- `'r'` : Opens a file in read-only mode (no modification possible).

In this example, the `flush()` method ensures that the written line is immediately saved to disk, even if the file is not yet closed.

```
# Open a file in write mode
file = open("data.txt", "w")

# Write data to the file
file.write("Writing important data.\n")

# Force data to be saved to disk
file.flush()

# Close the file
file.close()
```

### Writing Data to an SD Card

It is also possible to write data to a separate SD card connected to the Raspberry Pi Pico via SPI. This requires uploading the `sdcard` library to the board and importing it at the beginning of the program, as well as importing the `os` library and the `Pin` and `SPI` classes from the `machine` library. Then, you need to initialize communication with the card and mount it on the Raspberry Pi Pico's file system. The `sdcard` library and an example are provided in the course resources.

### Help, I deleted my card data!

If you accidentally deleted the flight data from your card, for example by deleting the file with Thonny or if you opened the file with the Python program in write mode instead of append mode, it is still possible to recover the data. Indeed, as on a computer, when a file is deleted from the Pico, the file's data is still present in memory, only the file index is deleted. If the card has not been reused after the data deletion, there is a high chance that the data has not been overwritten and can be recovered.

To do this, you need a Linux computer (or a simple virtual machine like Lubuntu) and a second Raspberry Pi Pico to serve as a debugger, along with a few cables to connect the Pico from which you want to recover data to the debugger and the picoprobe.uf2 file.

To configure the debugger, press the BOOTSEL button on the Pico that will serve as the debugger and connect it to the computer while holding down the button. Release the button. A file explorer for the Pico opens, drag and drop the picoprobe.uf2 file into it. The Pico is now configured as a debugger.

To recover data with Linux, open a terminal and install openocd with the following command.

```
sudo apt install openocd
```

Navigate to the directory where you want to recover the memory of the card to be debugged. If needed, create a directory with the mkdir command, then go to it with the cd command. Connect the debugger to the card to be debugged as follows.

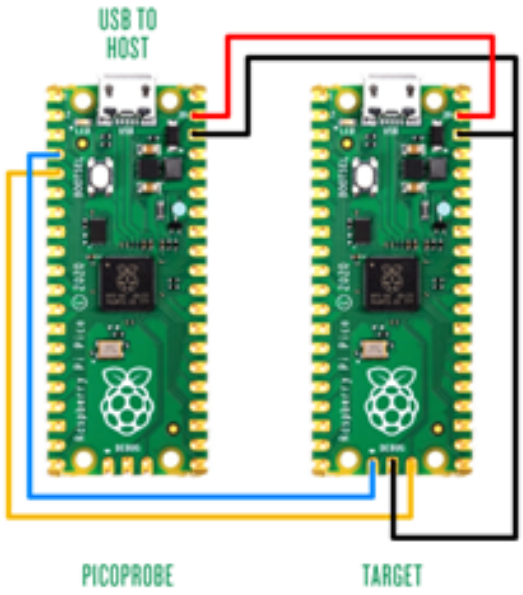



Figure 6: Connecting the debugger to the card to be debugged

While keeping the cards well connected and steady, execute the following command.

```
sudo openocd -f interface/cmsis-dap.cfg -f target/
rp2040.cfg -c "adapter speed 5000" -c "init" -c
"halt" -c "flash read_bank 0 dump.bin" -c "exit"
```

The memory of the Raspberry being debugged is copied into the dump.bin file. Exit the terminal and rename the dump.bin file to dump.txt. Now, simply locate the desired data in the dump.txt file!