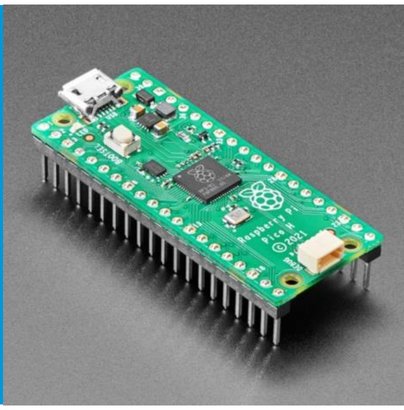
Belgium esero

Student workbook

CanSat 2022-2023

Workbook for the new Python kit



This workbook, written together with ESERO Luxembourg, presents the new CanSat kit based on Raspberry Pi Pico microcontroller board and guides the teams through the process of assembling their kit and coding it in Python.

esa belspo La Scientothèque L'ÉGALITÉ DES CHANCES PAR LES SCIENCES

CanSat 2022 Workbook

Table of content

Introduction

This document is to the point without being a complete technical guide. References to full technical guides are given where necessary.

The difficulty of the labs is progressive, starting with wiring and programming steps without any soldering.

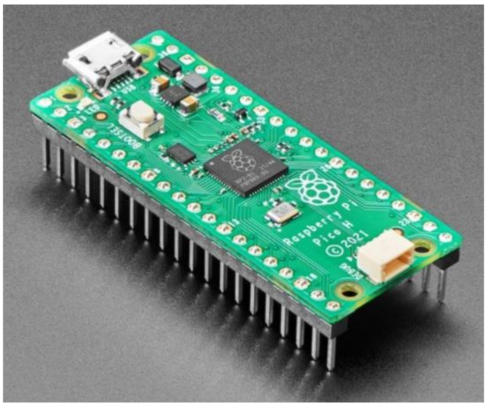
The assembly of components and their soldering only comes once the elements have been validated on the breadboard.

The following labs have been designed to introduce you to some of the electronics and programming skills that will be required to undertake the CanSat competition.

The CanSat Kit

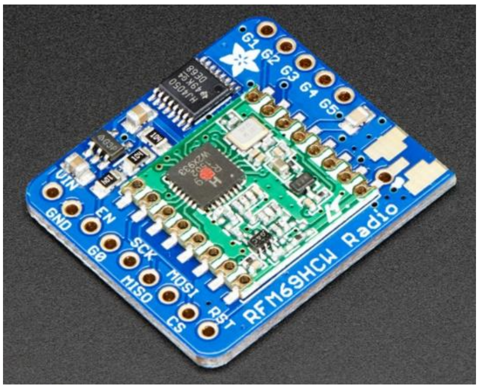
The kit comprises of off-the-shelf hardware that is cheap and easy to buy online. This allows teams to easily replace broken components and to also find support and ideas from the wealth of online teaching tutorials and technical resources related to Raspberry Pi Pico and MicroPython. The kit we use in the labs contains the following items:

2x Raspberry Pi Pico



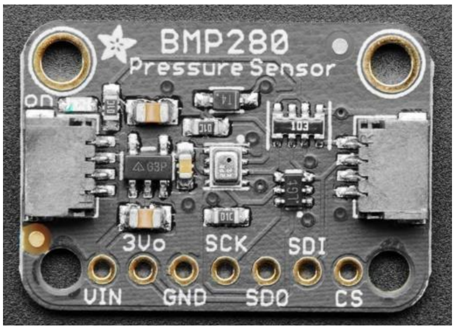
https://www.adafruit.com/product/5525

2x radio transceivers RFM69HCW



https://www.adafruit.com/product/3071

1x BMP280 Barometric Pressure/Temperature/Altitude Sensor



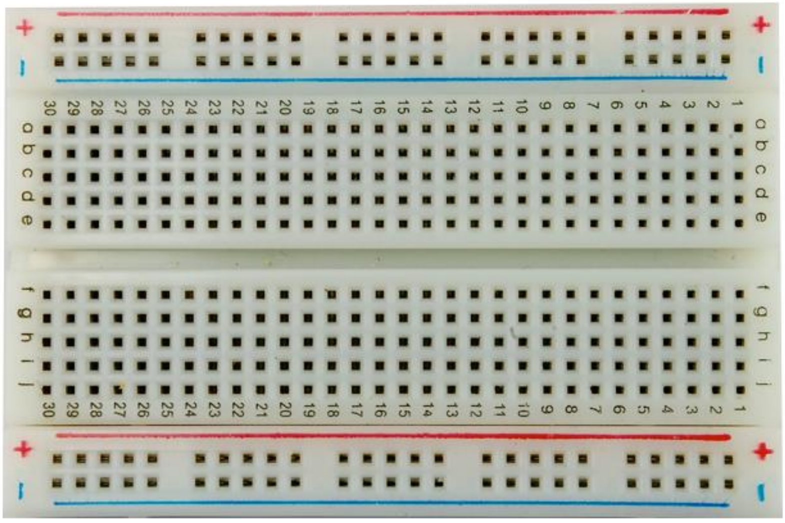
https://www.adafruit.com/product/2651

1x TMP36 sensor

Alternative to the BMP280, the TMP36 is a basic analogue temperature sensor that outputs a voltage based on the ambient temperature around the sensor.

https://learn.adafruit.com/tmp36-temperature-sensor

1x breadboard for prototyping circuits



A breadboard is a construction base used to build semi-permanent prototypes of electronic circuits without any soldering.

1x USB cable

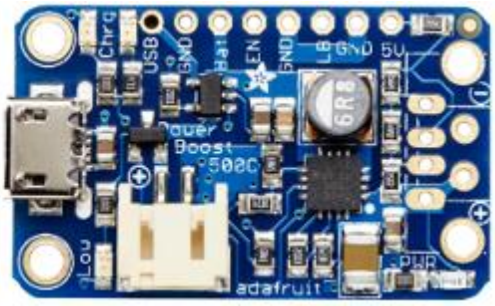
Cable used to plug your Raspberry Pico to a computer to program it or to charge the Lipo lithium battery.

1x lithium battery

3.7V 1300mAh battery to power the Raspberry Pico inside your CanSat, without the USB cable.

https://cdn-shop.adafruit.com/datasheets/Li-poly+603562-1300mAh.pdf

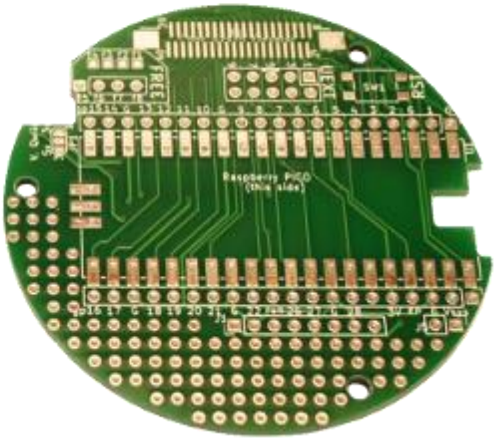
1x 5 volts converter and lithium battery charger



Board that converts the lithium battery power to a 5 volts power source suitable for the Raspberry Pico. It can recharge the lithium battery when the kit is powered via USB.

https://www.adafruit.com/product/1944

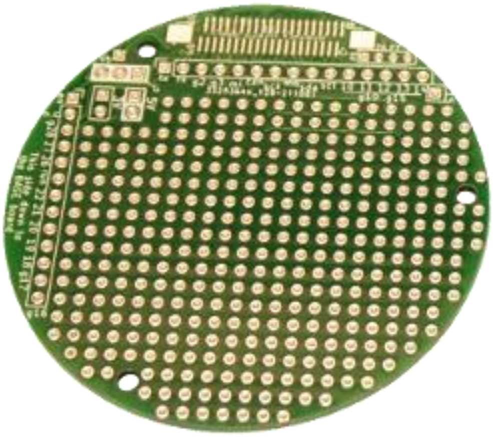
1x Cansat base board for Pico



After having tested the wiring of your components and their related software on the breadboard, you can solder the Raspberry Pico, the 5V converter board and the radio transmitter to this board. Its diameter fits within the maximal CanSat diameter. The BMP280 board can be plugged in using the provided JST cable.

https://shop.mchobby.be/fr/pico-rp2040/2275-carte-de-base-cansat-pour-pico-3232100022751.html

1x Cansat extension board



This CanSat extension board is useful to add components needed for the secondary mission like a GPS or other sensors.

https://shop.mchobby.be/fr/pico-rp2040/2272-carte-de-prototypage-cansat-pour-pico-3232100022720.html

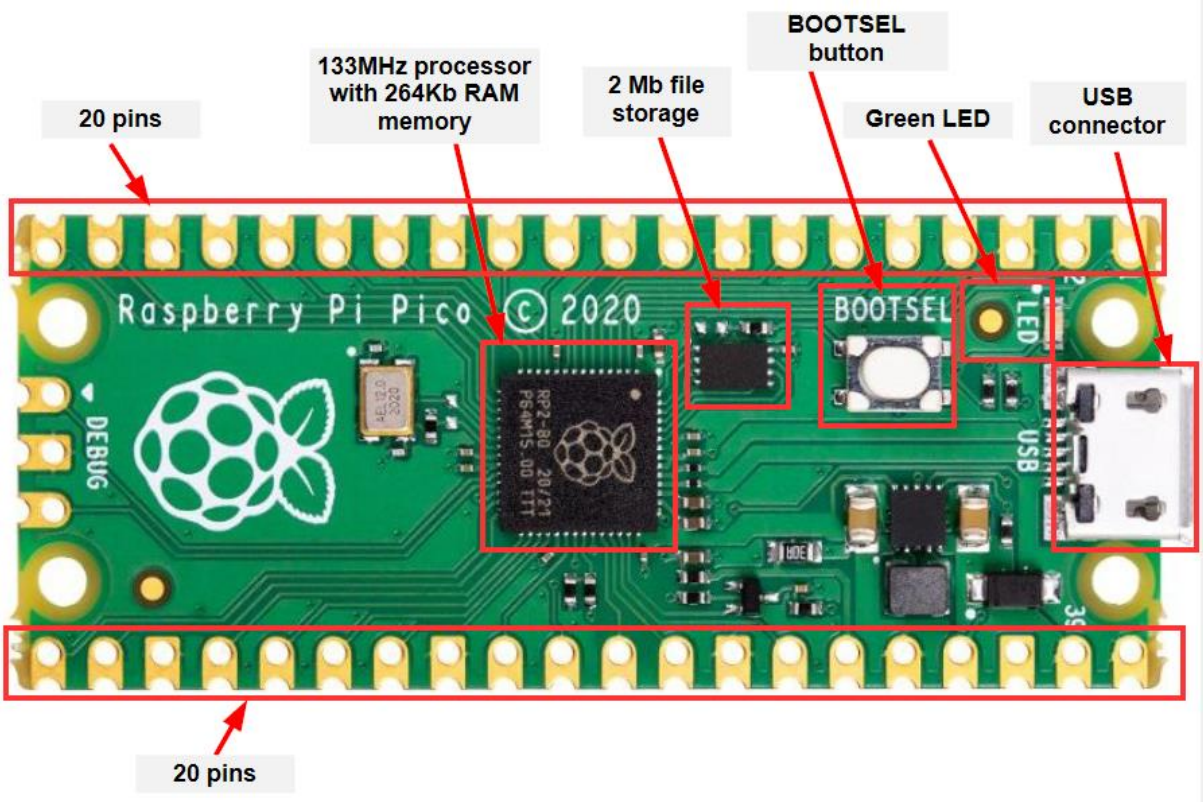
ESERO provides you with a complete kit for free but the various components can be bought separately at McHobby

Meet Raspberry Pi Pico

A Raspberry Pi Pico is a low-cost microcontroller device. Microcontrollers are tiny computers, but they only have a small file storage (unlike a hard drive on a typical computer) and lack peripheral devices that you can plug in (for example, keyboards or monitors).

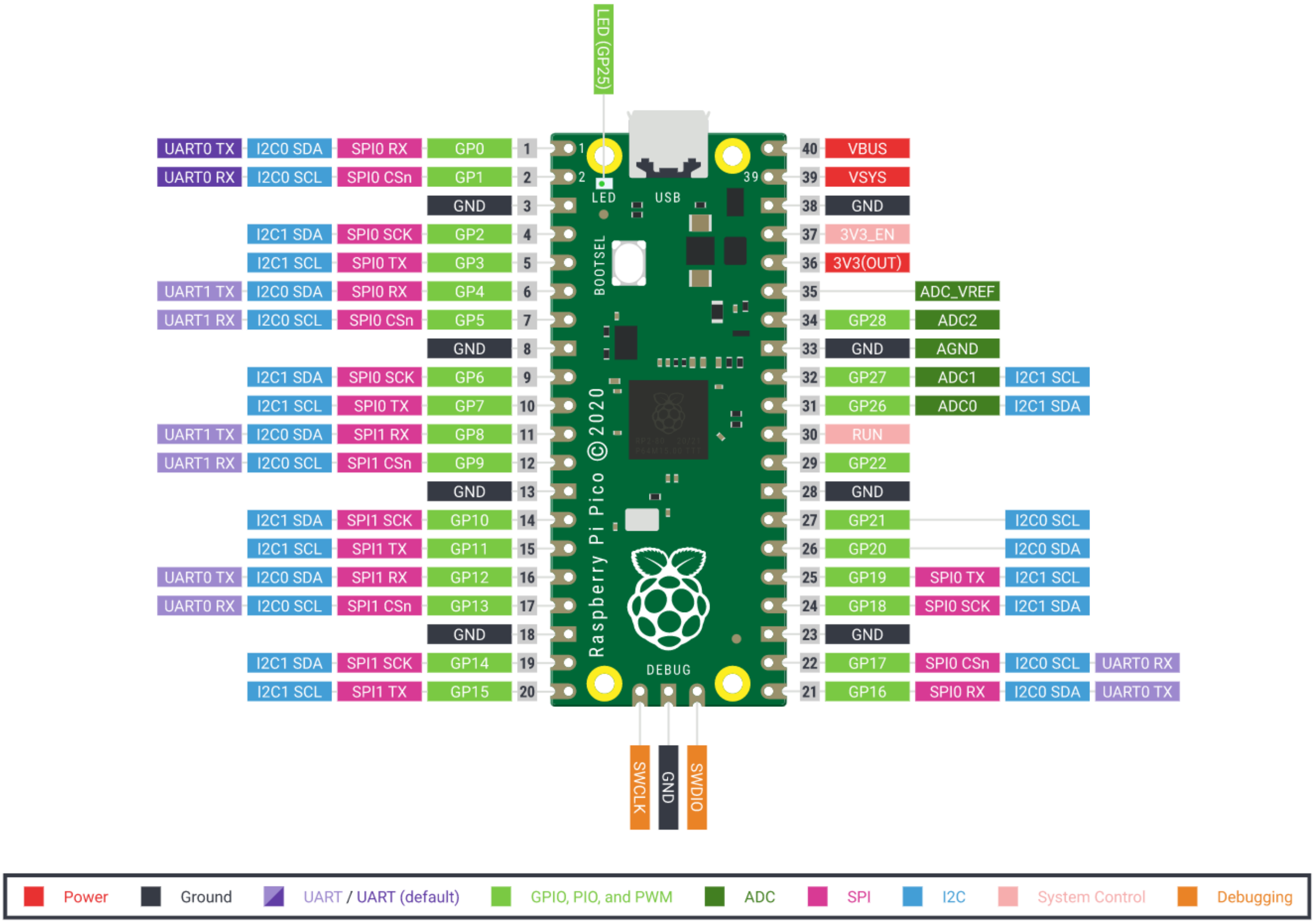
A Raspberry Pi Pico has

- A 133MHz processor with 264 kilobytes of RAM memory
- 2 megabytes of file storage
- A BOOTSEL button used to install MycroPython on the Pico
- A green LED
- A USB connector to power the Pico and transfer software or data.
- 2 x 20 pins used to power the Pico as well as control and receive input from a variety of electronic devices.



Each of the 40 pins has its own function.

If you need to know the pin numbers for a Raspberry Pi Pico, you can refer to the following diagram or this interactive website



While working with the Pico, you will only need to work with:

- Red and black pins = pins related to power and ground connection
- Purple pins = pins related to the UART communication
- Rose pins = pins related to the SPI communication protocol
- Blue pins = pins related to the I2C communication protocol

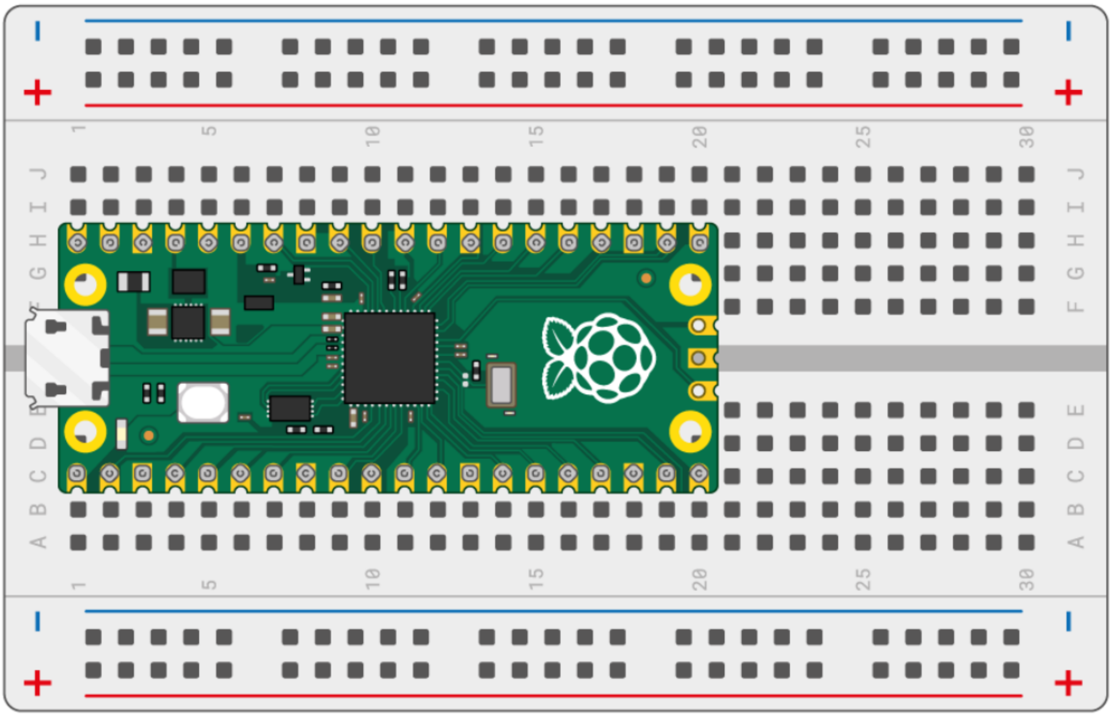
There are several ways to power your Pico, either using

- the micro-USB cable (in the development phase)
- the provided 5 volts converter and lithium battery (when development is done)

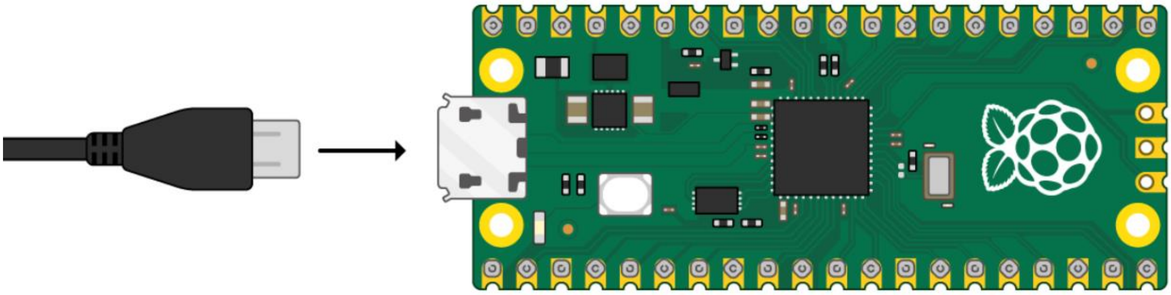
Connect your Raspberry Pi Pico to your computer

In this section, you will connect a Raspberry Pi Pico to another computer and learn how to program it using MicroPython.

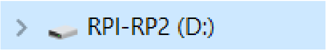
Firmly plug your Raspberry Pi Pico on the provided breadboard like shown in the following picture. Place it so that it is separated by the breadboard's ravine in the middle.



Plug the provided micro-USB cable into the port on the left-hand side of the Pico.



Your Pico should appear like an USB storage on your file system



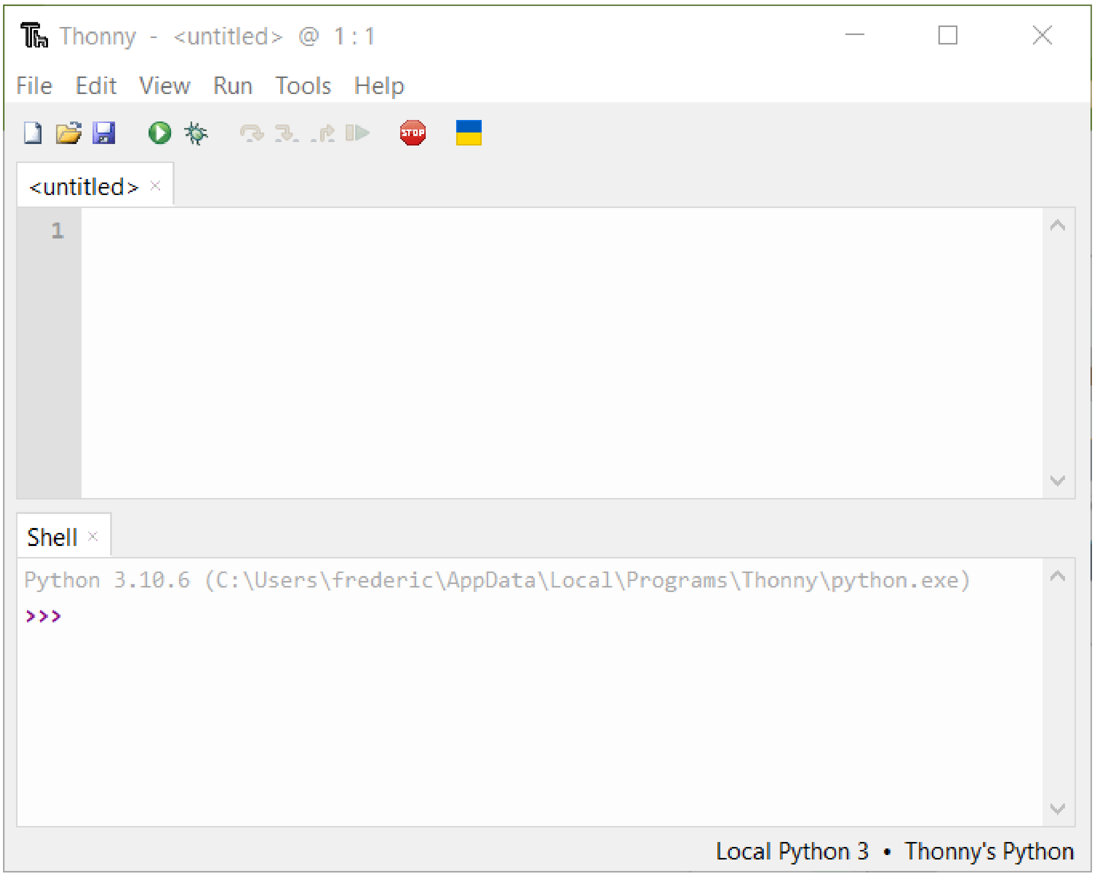
RPI-RP2 (D:)

Install Thonny on your computer

Python is a general purpose language used in a large variety of applications (data sciences, Artificial Intelligence, statistics, …) while MicroPython is specifically designed for microcontrollers like the Raspberry Pi Pico used in our project.

To edit, run and debug our code in MicroPython language we will install an Integrated Development Environment (IDE) called Thonny, available at https://thonny.org

Open Thonny from your application launcher. It should look something like this:

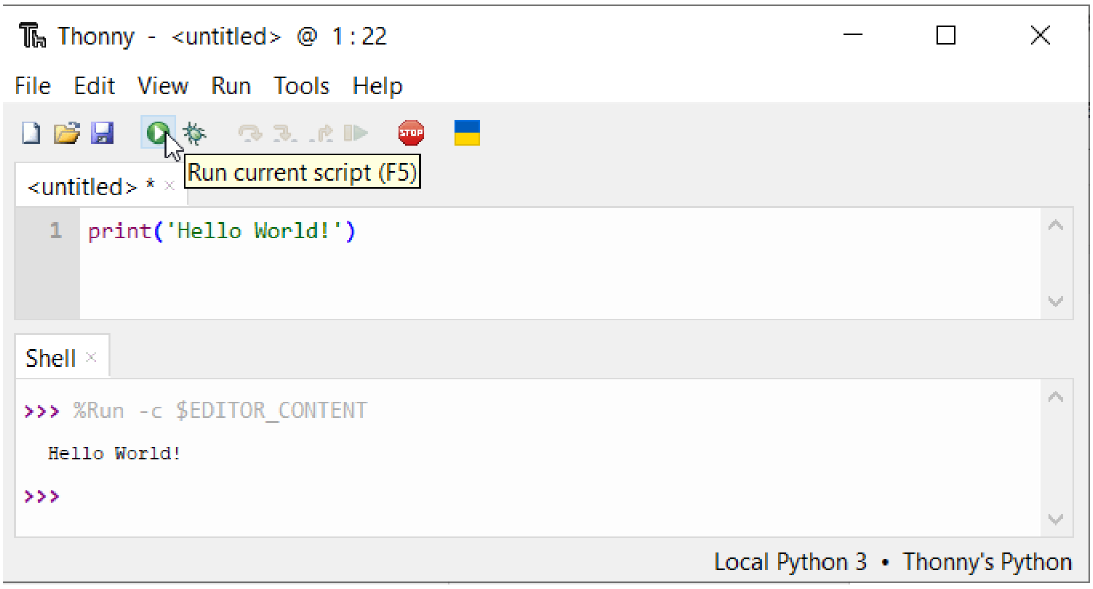


```
Python 3.10.6 (C:\Users\frederic\AppData\Local\Programs\Thonny\python.exe)
>>>
```

You can use Thonny to write standard Python code. Type the following in the top window, and then click the Run button.

```
print('Hello World!')
```

The result is shown in the “Shell” window


Thonny - <untitled> @ 1 : 22
File Edit View Run Tools Help
Run current script (F5)
<untitled> *
Shell
Local Python 3 • Thonny's Python

```
print('Hello World!')
```

```
>>> %Run -c $EDITOR_CONTENT
  Hello World!
>>>
```

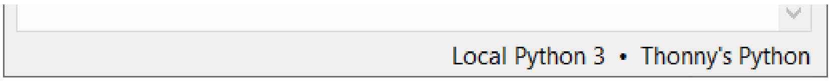
Install MicroPython on your Pico

Your new Raspberry Pi Pico needs MicroPython to run your software.

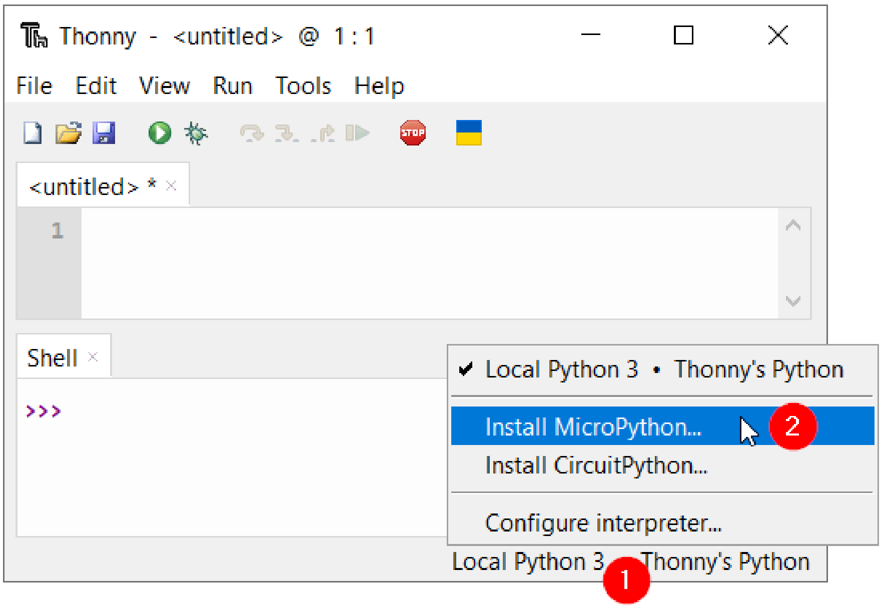
Start by unplugging the micro-USB cable from your computer but leave it connected to your Pico.

Press the BOOTSEL button and hold it while you connect the other end of the micro-USB cable to your computer.

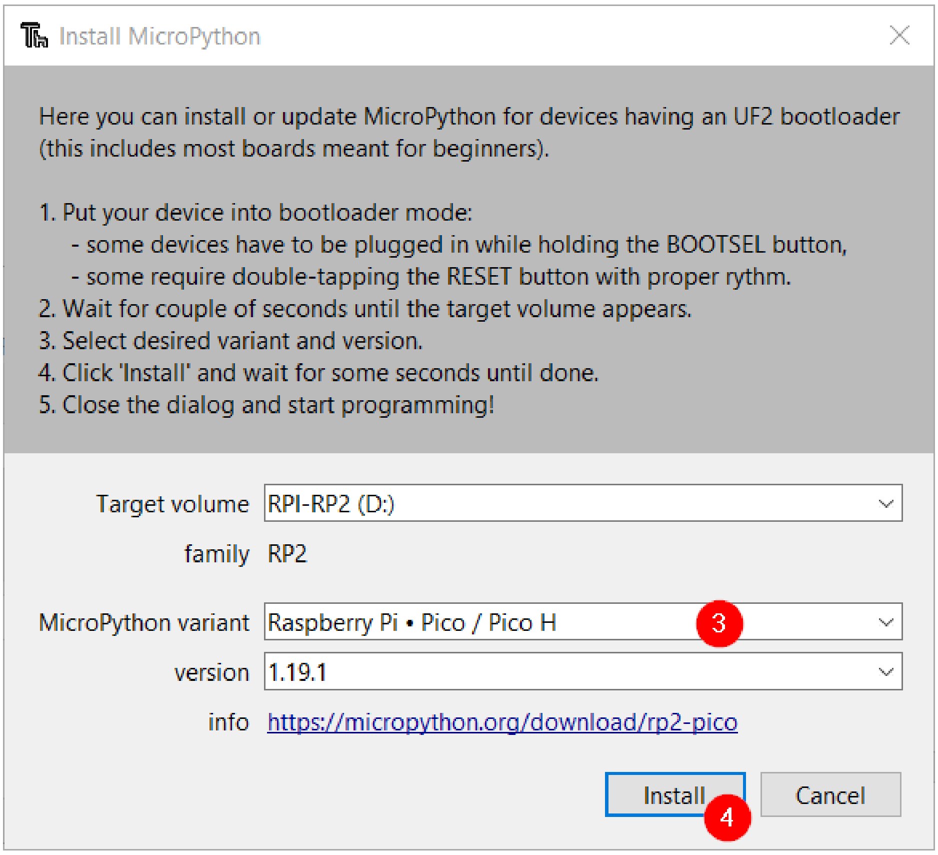
In the bottom right-hand corner of the Thonny window, you will see the version of Python that you are currently using.



Left-click on the Python version and choose 'Install MicroPython…'


Thonny - <untitled> @ 1 : 1
File Edit View Run Tools Help
<untitled> *
Shell
>>>
✔ Local Python 3 • Thonny's Python
Install MicroPython...
2
Install CircuitPython...
Configure interpreter...
Local Python 3 • Thonny's Python
1

A dialog box will pop up to install the MicroPython firmware on your Pico. Select the correct MycroPython variant and click on the Install button.



Here you can install or update MicroPython for devices having an UF2 bootloader (this includes most boards meant for beginners).

1. Put your device into bootloader mode:
   - some devices have to be plugged in while holding the BOOTSEL button,
   - some require double-tapping the RESET button with proper rythm.
2. Wait for couple of seconds until the target volume appears.
3. Select desired variant and version.
4. Click 'Install' and wait for some seconds until done.
5. Close the dialog and start programming!

Target volume: RPI-RP2 (D:)

family: RP2

MicroPython variant: Raspberry Pi • Pico / Pico H

version: 1.19.1

info: https://micropython.org/download/rp2-pico

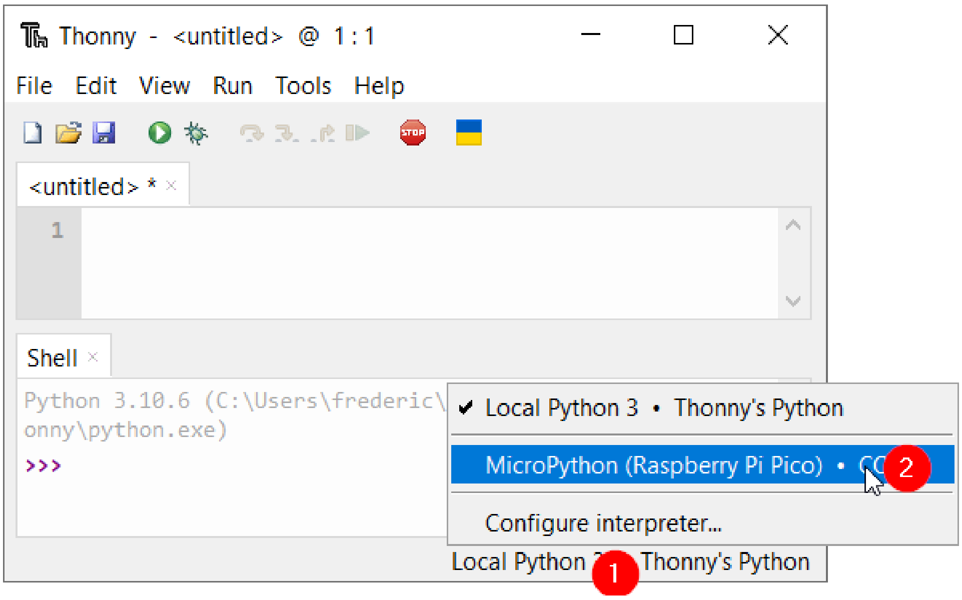
Wait for the installation to complete and click on the Close button.

You don’t need to update the firmware every time you use your Pico.

Next time, you can just plug it into your computer without pressing the BOOTSEL button.

Run your first MicroPython code on your Pico

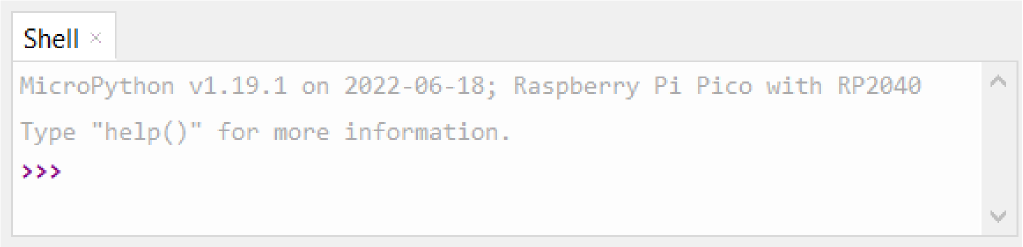
Make sure that your Raspberry Pi Pico is still connected to your computer.
Select the MicroPython (Raspberry Pi Pico) interpreter on the bottom right.


Thonny - <untitled> @ 1 : 1
File
Edit
View
Run
Tools
Help
<untitled> *
1
Shell
Local Python 3 • Thonny's Python
MicroPython (Raspberry Pi Pico) • CO
2
Configure interpreter...
Local Python 3 • Thonny's Python
1

```
Python 3.10.6 (C:\Users\frederic\
onny\python.exe)
>>>
```

Look at the Shell panel at the bottom of the Thonny editor.

You should see something like this:



```
MicroPython v1.19.1 on 2022-06-18; Raspberry Pi Pico with RP2040
Type "help()" for more information.
>>>
```

Thonny is now able to communicate with your Pico using the REPL (read–eval–print loop), which allows you to type Python code into the Shell and directly see the result.

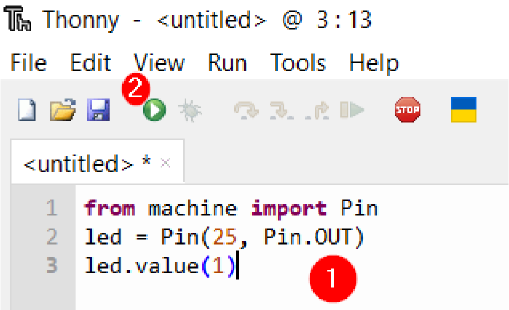
MicroPython adds hardware-specific modules, such as `machine`, that you can use to program your Raspberry Pi Pico.

Let’s create a `machine.Pin` object to play with the onboard LED, which can be accessed using GPIO pin 25.

If you set the value of the LED to `1`, the onboard LED turns on.

Enter the following code, make sure you tap Enter after each line.

```
from machine import Pin
led = Pin(25, Pin.OUT)
led.value(1)
```



```
from machine import Pin
led = Pin(25, Pin.OUT)
led.value(1)
```

After pressing the green “Run” button, you should see the onboard LED light up.

Change the code and set the LED value to 0 to turn the LED off.

```
led.value(0)
```

Turn the LED on and off as many times as you like.

Tip: You can use the up arrow on the keyboard to quickly access previous lines.

We said the onboard LEP is connected to GPIO pin 25, but what is GPIO?

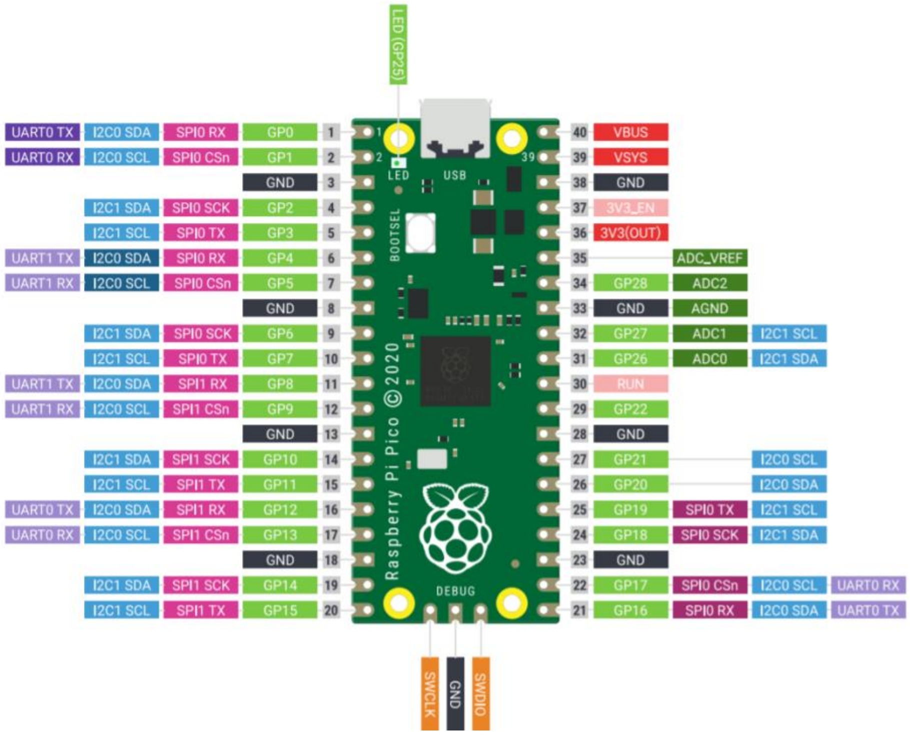
General Purpose Input/Output (GPIO)

GPIO pins on the Raspberry Pi allow external voltages to be read from the software and they also allow external voltages to be set from software. These are digital pins, so the inputs are interpreted at either a logical "False" or logical "True" depending on the voltage of the signal.

For our 3.3V Raspberry Pi, any voltage under 2.5V is interpreted as "False" and conversely any voltage over 2.5V is interpreted as true (up to 3.3V). This is similar for output signals. A "True" output will set the pin's voltage to 3.3V and the "False" output will set the pin's voltage to 0V.

GPIO pins can be used as either input or output ports and this set by software.

The Pi Pico has 28 GPIO ports as seen in the green boxes in the following diagram. Many pins are multi-purpose and can also be used for other interfaces (UART, SPI, I2C), these are represented by the multi-coloured boxes to the side of the green boxes in the diagram. The following link contains the pinout: https://datasheets.raspberrypi.org/pico/Pico-R3-A4-Pinout.pdf



https://datasheets.raspberrypi.org/pico/Pico-R3-A4-Pinout.pdf

Typical GPIO CanSat Uses:
Inputs: On/Off based sensors, switches, buttons, deployment sensors
Outputs: Status LEDs, basic servos (PWM is better), turning sensors on, resetting sensors connected by other signals

If you want to write a longer program, then it is best to save it in a file. You will do this in the next section.

Write your first microPython file to control the onboard LED

The Shell is useful to try out quick commands. However, it is better to put longer programs in a file.

Thonny can save and run MicroPython programs directly on your Raspberry Pi Pico.

In this step, you will create a MicroPython program to blink the onboard LED on and off in a loop, using GPIO pins

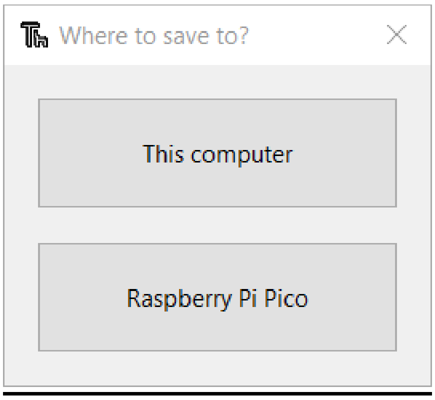
Go back to Thonny and click in the main editor pane of.

Enter the following code to toggle the LED.

```
# import necessary pre-existing libraries
from machine import Pin
# Declare a variable named “led”, link it to pin number 25 and define it as output
led = Pin(25, Pin.OUT)
# Change the led state from led.value(0) to led.value(1) and vice versa
led.toggle()
```

The complete “Pin” library documentation can be found in the official MicroPython documentation

Click the **Save** button to save your code and the following screen will show up:



Choose “Raspberry Pi Pico” and name the file “blink.py”

Tip: You need to enter the .py file extension so that Thonny recognises the file as a Python file. Thonny can save your program to your Raspberry Pi Pico and run it.

You should see the onboard LED switch between on and off each time you click the Run button.

But what if you want to see the LED blinking without having to click the Run button over and over ?

To achieve this we will use a “while” loop and the “sleep” function

Be careful to indent the code with 4 spaces within the while loop to let MicroPython know that these lines are part of the while loop.

Update your code so it looks like this:

```
# import necessary pre-existing libraries
from machine import Pin
From time import sleep
# Declare a variable named “led”, link it to pin number 25 and define it as output
led = Pin(25, Pin.OUT)
# Change the PIN state to HIGH if it was LOW and vice versa, therefore cutting or
supplying the power to the LED

# Be careful to indent the code with 4 spaces within the while loop to let
MicroPython know which lines are part of the while loop.
while True:
    # Change the led state from led.value(0) to led.value(1) and vice versa
    led.toggle()
    # Halt the program execution for half a second
    sleep(0.5)
```

You can also use the Timer module (first line below) to set a timer that runs a function at regular intervals. Update your code so it looks like this:

```
# import necessary pre-existing libraries
from machine import Pin, Timer

# Declare a variable named “led”, link it to pin number 25 and define it as output
led = Pin(25, Pin.OUT)
# Declare a timer variable to deal with timing of periods and events
timer = Timer()

# Declare a function named “blink” that toggle the LED state
def blink(timer):
    # # Change the led state from led.value(0) to led.value(1) and vice versa
    led.toggle()

# Configure the timer to call the pre-defined blink function every 2.5 seconds
timer.init(freq=2.5, mode=Timer.PERIODIC, callback=blink)
```

The complete “Timer” library documentation can be found in the official MicroPython documentation

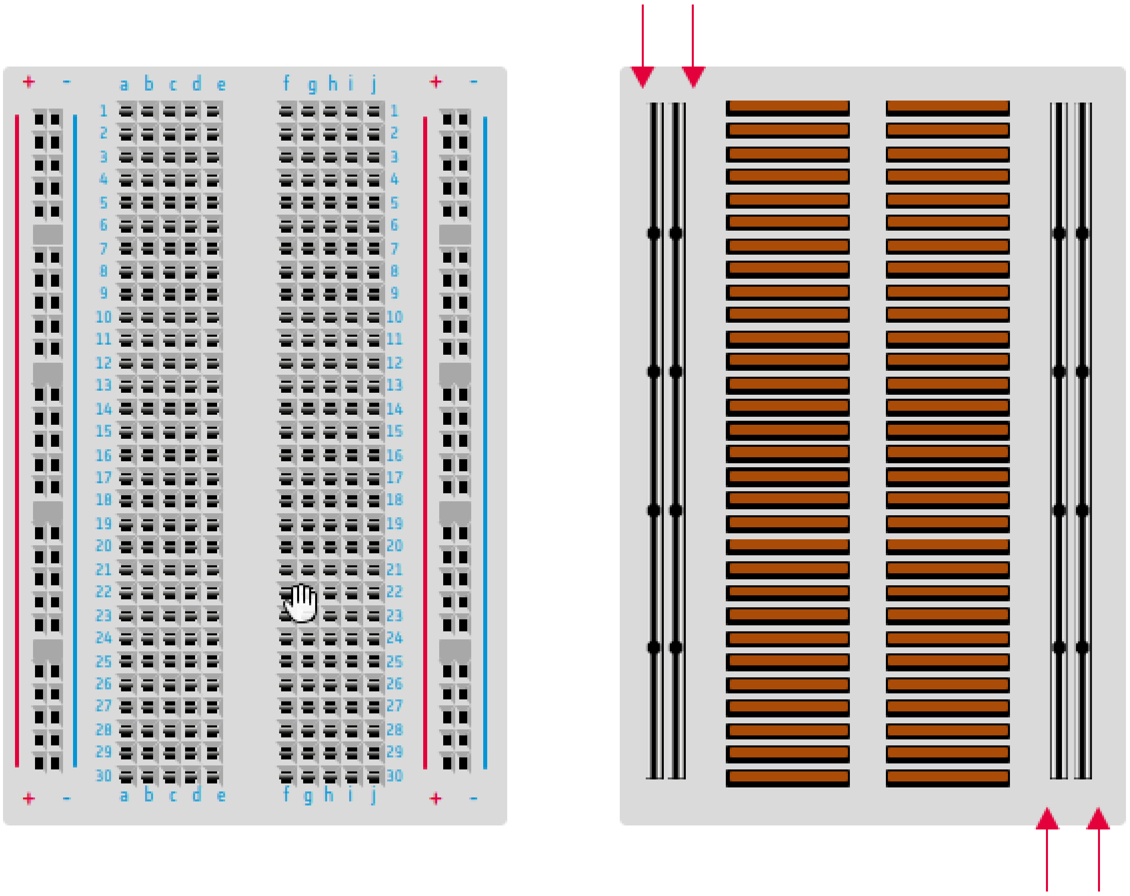
Click **Run** and your program will blink the LED on and off until you click the **Stop** button.

Tip: If you unplug/re-plug your Pico, you will have to press again the **Run** button to run the program. But if you name your program “**main.py**” on your Pico it will run automatically when the Pico powers up.

Software development & wiring test

Meet a breadboard

Whilst you are learning the basics of Pico and sensors it is best to use a solderless breadboard, as any mistakes you make building your circuit can be easily changed.
A breadboard is a simple tool that can be used to wire electrical components together.



Pins of electrical components can be placed into the terminals on the board. Centrally, rows are horizontally connected. This means for example, that the two pins of a resistor should be placed in different rows, otherwise it will form a closed circuit with itself.

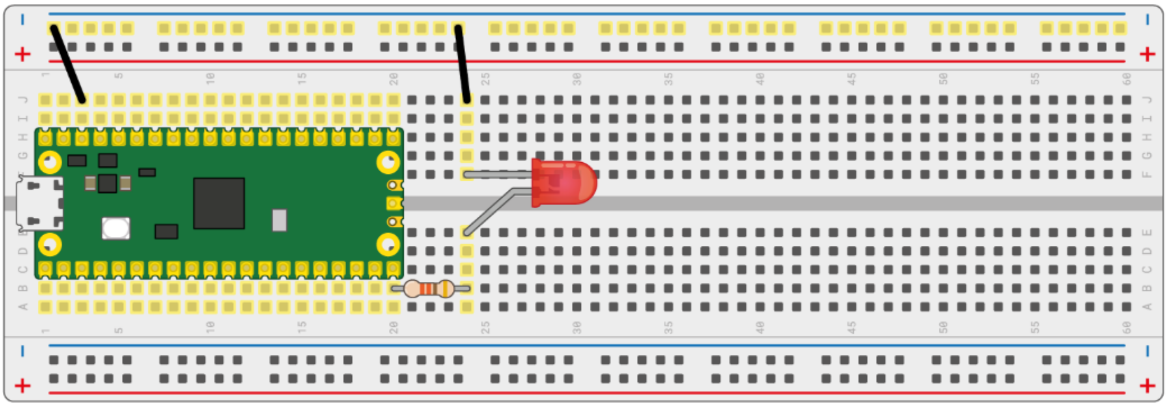
It is very important to make a sketch of your circuit before connecting and powering the circuit, because you will risk breaking the components. The outer columns of the board are connected in columns, rather than rows. Typically, these are used to provide ground and voltage connections.

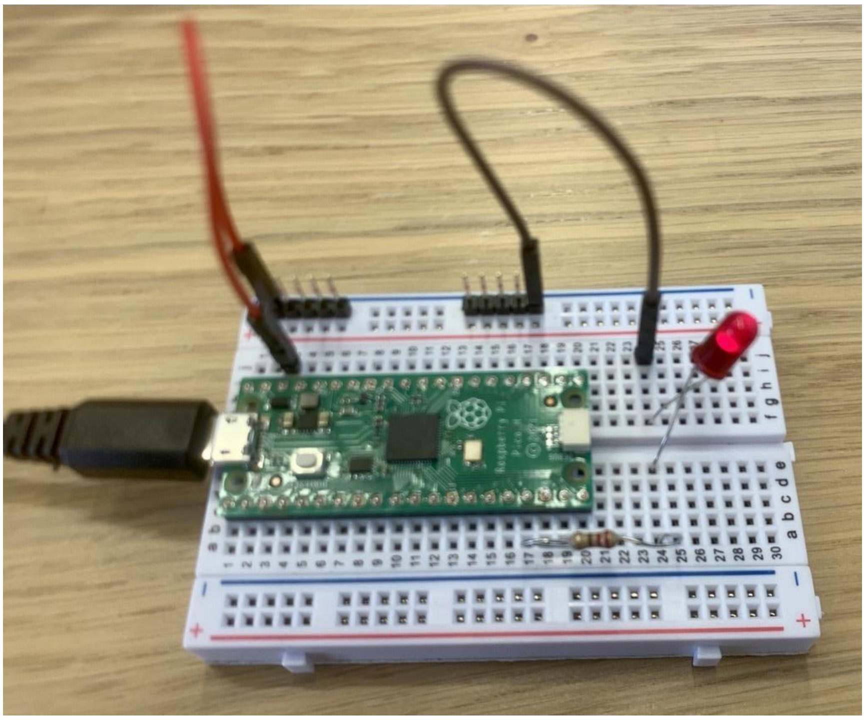
Blink a LED on the breadboard

You will know learn how to control an external LED.

Use a 220 ohms resistor, an LED, some female jumpers and a few headers to connect up your Raspberry Pi Pico on your breadboard as shown in the image below.

Note how the LED is connected on GPIO 15 on one side (the last one on the bottom left as you can see in the Pico pinout diagram) and to the Pico's ground pin on the other side.





In Thonny, reuse the code from the previous section, but instead of GPIO 25, use GPIO 15

```
….
# Declare a variable named “led”, link it to pin number 15 and define it as output
led = Pin(15, Pin.OUT)
….
```

In this example, we chose to connect the LED to GPIO 15 but if you can use another GPIO if you want.

Test your jumpers

Your kit contains several male and female jumpers to test your components wirings on your breadboard before soldering your components on the CanSat base board.

Sometimes, because of factory issues, it happens that some if these jumpers are broken.

To save you some headaches and time, we strongly advise you to test all your jumpers with the previous “blink an external LED” exercise by replacing the 2 jumpers with the other jumpers provided with the kit.

Measuring temperature and pressure

Micro Python provides some built in functionality for managing the Pi Pico, however this can be extended through the use of third-party libraries. These are libraries produced by manufacturers, suppliers and the Micro Python community for the purpose of using extra devices with the Pi Pico. These libraries reduce the complexity of using external devices by providing high-level functions to interact with the devices they support

The kit is made of several sensors for which we will use specific Micro Python libraries we can find on the WEB.

We will install the libraries which have been recommended and tested by the supplier.

Install BMP280, RFM69 Boards libraries

BMP280

This chip measures the atmospheric pressure as well as the temperature.

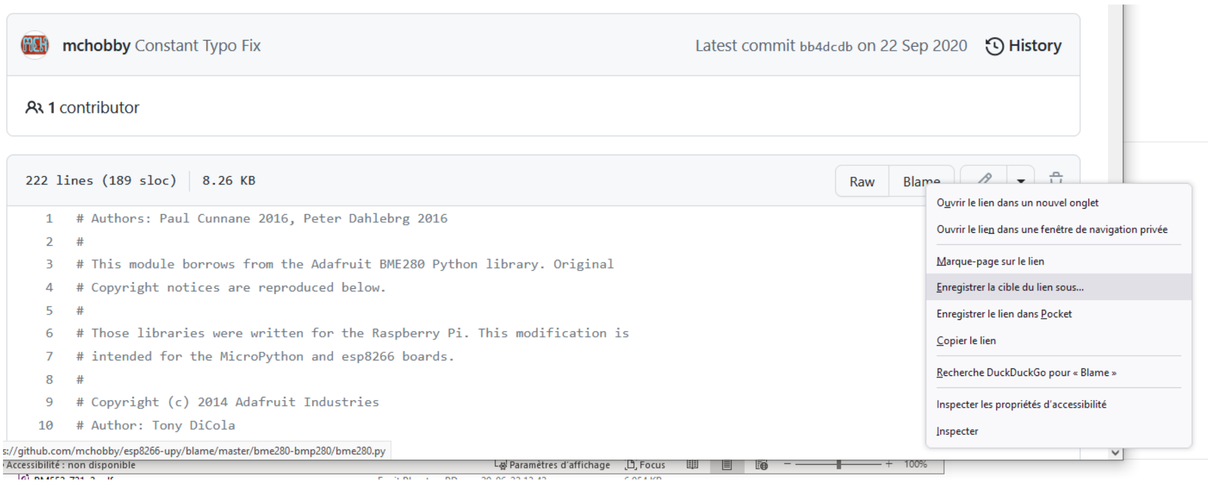
Click on this link to access and download the BMP280 – BME280 library :

https://github.com/mchobby/esp8266-upy/blob/master/bme280-bmp280/bme280.py

then right click the RAW button

choose enregistrer la cible du lien sous

keep the proposed file name (bmp280.py) and change the destination directory if you want (default is download)



mchobby Constant Typo Fix   Latest commit bb4dcdb on 22 Sep 2020

1 contributor

222 lines (189 sloc) | 8.26 KB

```
# Authors: Paul Cunnane 2016, Peter Dahlebrg 2016
#
# This module borrows from the Adafruit BME280 Python library. Original
# Copyright notices are reproduced below.
#
# Those libraries were written for the Raspberry Pi. This modification is
# intended for the MicroPython and esp8266 boards.
#
# Copyright (c) 2014 Adafruit Industries
# Author: Tony DiCola
```

s://github.com/mchobby/esp8266-upy/blame/master/bme280-bmp280/bme280.py

History
Raw
Blame
Ouvrir le lien dans un nouvel onglet
Ouvrir le lien dans une fenêtre de navigation privée
Marque-page sur le lien
Enregistrer la cible du lien sous...
Enregistrer le lien dans Pocket
Copier le lien
Recherche DuckDuckGo pour « Blame »
Inspecter les propriétés d'accessibilité
Inspecter
Accessibilité : non disponible
Paramètres d'affichage
Focus
100%

RFM69HCW

This chip is the radio receiver/sender installed on the base station and in the CatSat station to transmit the data from the CatSat Station to the Base Sation:

Click on this link to access and download the RFM69HCW library :

https://github.com/mchobby/esp8266-upy/blob/master/rfm69/lib/rfm69.py

… proceed in the same way as for the previous chip

Transfer the bme280.py and rfm69.py on the Pi pico

Both programs have to be transferred from the downloads directory on your PC (Windows, Raspi or Mac), to the Raspberry Pi Pico /lib directory as they will be used by the complete program for the CatSat mission

This is quite simple to do this:

Click on the view button in the menu bar

Choose the Files option

On the left side the Files view shows the directory on the PC and of the Pi pico (if the Pi pico is connected, you see the prompt >>> (if not there is no connected Pi pico)

In the file view, double click on the lib to enter this directory on the PI pico

Right click on the bme280.py in the downloads directory

1. Choose upload to /lib

Proceed in the same way to transfer the rfm69.py file from the PC/downloads direstory to the Pi p

Lab 1: GPIO Communication and Soldering

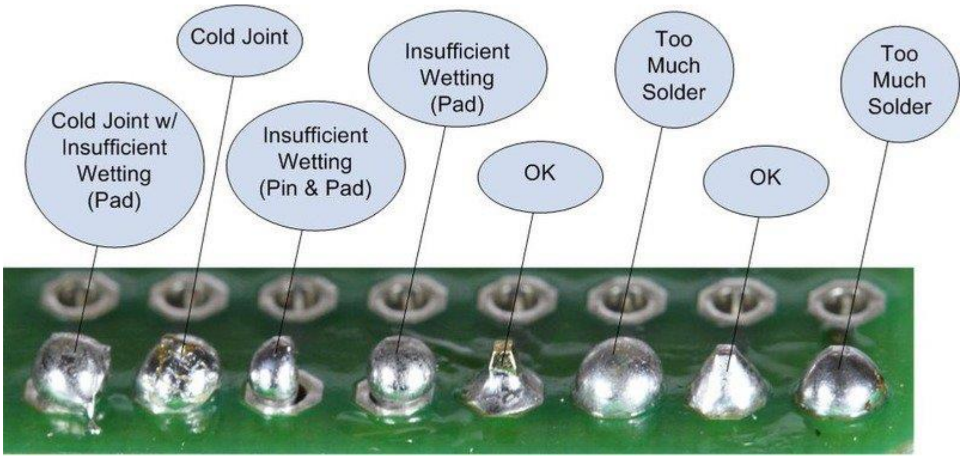
The first lab will cover running a basic **Python3** program that tests that MicroPython and the Pi Pico are up and running correctly. It also contains some soldering to build the boards in your kit. Soldered connections are one of the most reliable ways of connecting parts of the CanSat electrical design together.

Exercise 1.1: Soldering the CanSat Kits

The RFM96W, BMP280 and Raspberry Pi Pico boards will need header pins soldering to them so that they can be used with the breadboard and jumper wires.

For the RFM96W and BMP280 the boards can be soldered such that the long side of the header pins are facing the bottom of the boards. The Pi Pico on the other hand has its pin names on the back and so it may be preferable to solder these pins on backwards if you intend to use the Pi with a breadboard.

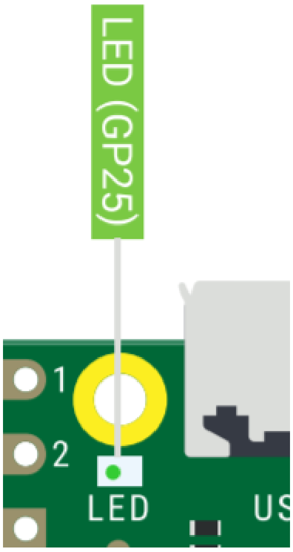
As the Pi pins are in parallel, it is recommended to plug them into a breadboard first to ensure they are aligned.


Cold Joint
Cold Joint w/ Insufficient Wetting (Pad)
Insufficient Wetting (Pin & Pad)
Insufficient Wetting (Pad)
OK
Too Much Solder
OK
Too Much Solder

https://learn.adafruit.com/adafruit-guide-excellent-soldering/common-problems

Exercise 1.2: Controlling the LED from the view Shell Using GPIO

The Pi Pico has a built in LED on GP25:



As this is a GPIO pin, we can control it from the MicroPython software. To test this will we use the *Read-Evaluate-Print-Loop (REPL)* functionality of Thonny thanks to the Shell view that allows us to write basic code without saving it to a file. The code has to be entered one line at a time, which can be tedious but is useful for testing.

1. Connect the Raspberry Pi Pico to the laptop.

2. Click the STOP icon

The following message should appear:

```
MicroPython v1.19.1 on 2022-06-18; Raspberry Pi Pico with
RP2040


Type "help()" for more information.

>>>
```

1.

We can write code directly into this interface. To test the LED we need to use the GPIO functions. To do this we need to import a MicroPython library.

>>> from machine import Pin

You have now a new line with the prompt >>> indicating that the Pin functions of the machine library can now be used

3.

Now we can set the LED as a GPIO pin with the following code:

```
>>> led = Pin(25,Pin.OUT)
```

1.

We can then control the LED from Python. The following line should turn on the LED:

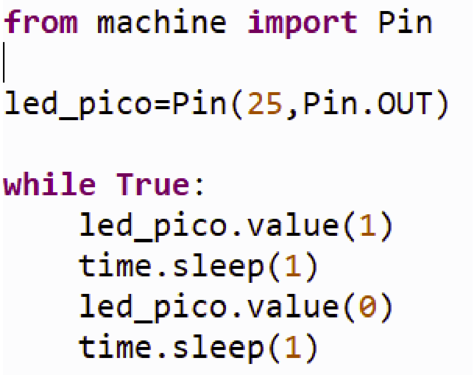
```
>>> led.value(1)
```

1.
write a command to switch the LED off !!

Exercise 1.3: Controlling the LED from Software Using GPIO

The MicroPython REPL (Shell View) is handy for testing small amounts of code, but for the CanSat application the code will need to be written into a file.

If this file is saved to the Pi Pico under the name main.py it will be automatically run when you're the Pi Pico is powered up. Any other file name will be run at startup of the Pi Pico



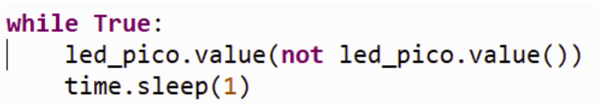
```
from machine import Pin

led_pico=Pin(25,Pin.OUT)

while True:
    led_pico.value(1)
    time.sleep(1)
    led_pico.value(0)
    time.sleep(1)
```

Note the indentation of this code. This is important in Python and shows what code should be executed as part of this loop.

1. Run the code. You will see some errors regarding the *sleep()* function. The sleep function is part of the "time" library. Therefore, add some code to import the *time* library in the same way as the machine library were added.

2. The code should now run.

3. Finally, we can simplify the above code by reading the state of the LED, inverting it (i.e. True -> False, False -> True) before writing to it back to the LED. Replace your while loop with the following code:



```
while True:
    led_pico.value(not led_pico.value())
    time.sleep(1)
```

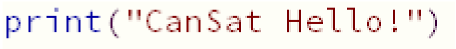
4. This *while* loop with the 1 second pause will be used later on in the workshop for reading the sensors once every second and sending the data over the radio.

Exercise 1.4: Printing Messages to the Console

MicroPython allows the Pi to output messages from the code, that can describe events and display variable values. This is very useful during the development of the CanSat code. There are many ways this can be achieved in MicroPython and we will show one method.

The messages are sent using a UART interface that is forwarded over the USB cable coming in the computer. Thonny will display these messages in the Shell View. You could also use a terminal emulator program (PuTTY, Tera Term for Windows) to connect to the serial port to display the messages. You can also use the UART interface to connect to other devices such as GPS receivers and GSM modems.

1. The *print()* function allows us to send messages over the UART. Add the following line before your while loop from the previous exercise:

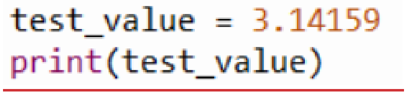


```
print("CanSat Hello!")
```

2. Run the code by saving the file and check that you can see the message in the serial window within the Shel view.

3. Information messages like this are useful for tracking the progress of the CanSat program and displaying error messages. However, they can also be used to print out variable values.

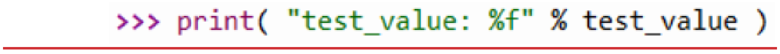
   Add the following code after the Hello message:



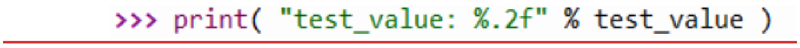
```
test_value = 3.14159
print(test_value)
```

4. Run the code. Python automatically converts the value of *test_value* into a message string.

5. A more useful case is to combine the text message and the value of a variable. Add and run the following code:



```
>>> print( "test_value: %f" % test_value )
```



```
>>> print( "test_value: %.2f" % test_value )
```

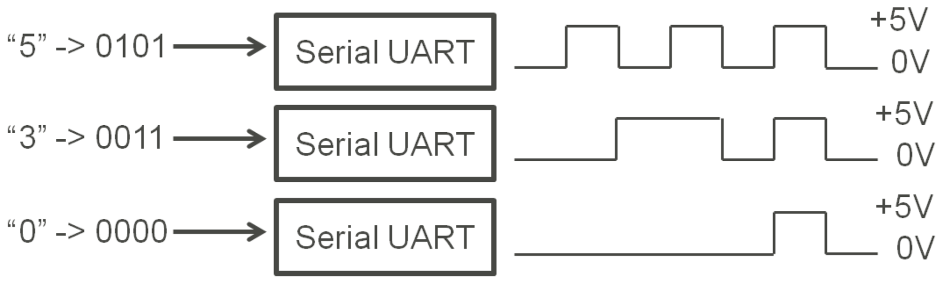
_The results are different. In the first example the printed value is the full value of the variable. In the second example we fixed the displayed number of the decimals at 2

The code after the ‘%’, within the double quote is a placeholder for a variable. The ‘I’ indicates the value is an integer, an f indicates it is a floating point variable. The "*:.2*" in the second example denotes that we want the displayed value with two decimals after the decimal point. Python displays the value of the variable which name is preceded by a ‘%’ after the second double quote.

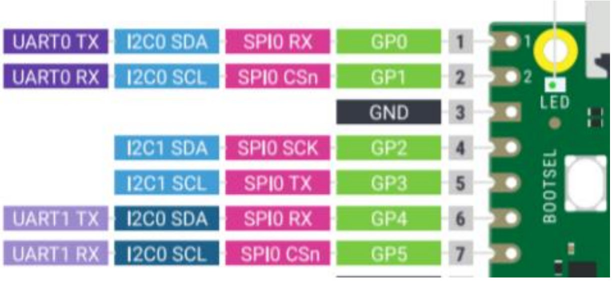
6.

Universal Asynchronous Receiver-Transmitter (UART)

Despite the name UART is a relatively simple communication interface. It operates in the same fashion as the GPIO with true/false values represented as 0V and 5V but pulses are sent across the wire instead of a steady voltage pulses. This allows a numerical value to be converted to a series of pulses and sent over a single wire:



The Raspberry Pi Pico has two UARTs. These can be connected to many pairs of GP pins as shown in purple in the Pi Zero pinout: TX is the transmit (i.e. data sent out of the Pi) and RX as the receive (i.e. data sent to the Pi).



Typical UART CanSat Uses:

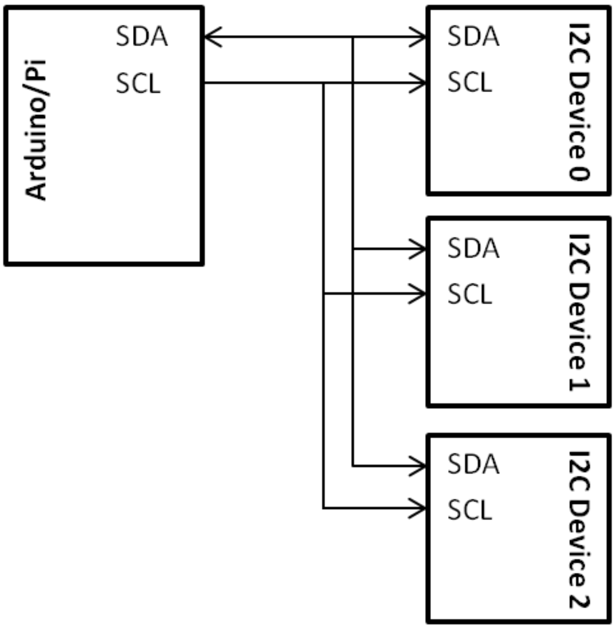
Sending debug and development messages to a PC, communicating with GPS sensors, communicating with external WiFi and GPRS (3G) modems.

Lab 2: Reading from Analogue and Digital Sensors (ADC and I2C)

This lab covers communication with the temperature and pressure sensor required to achieve the CanSat primary mission, using a more complex communication interface: I^{2}C.

I^{2}C Prototcol (Inter-Intergrated Circuit)

I2C allows multiple devices (up to 1008) to be connected to the same I2C interface with just a pair of wires. It also allows bi-directional communication over these two wires and so is ideal for communicating with many sensors. An example wiring with three devices would be as follows:



The software required to communicate with I2C devices can be complex, however most devices will have a software library provided that will give you functions that make the device easy to use. For example in this lab we use the provided BMP280 library to hide away the low-level I2C code.

Typical I2C CanSat Uses:
"Smarter" sensors (e.g. the BMP280), Accelerometers, Analog-Digital Converters, Digital-Analog Converters, LCD Screens, Battery Controllers

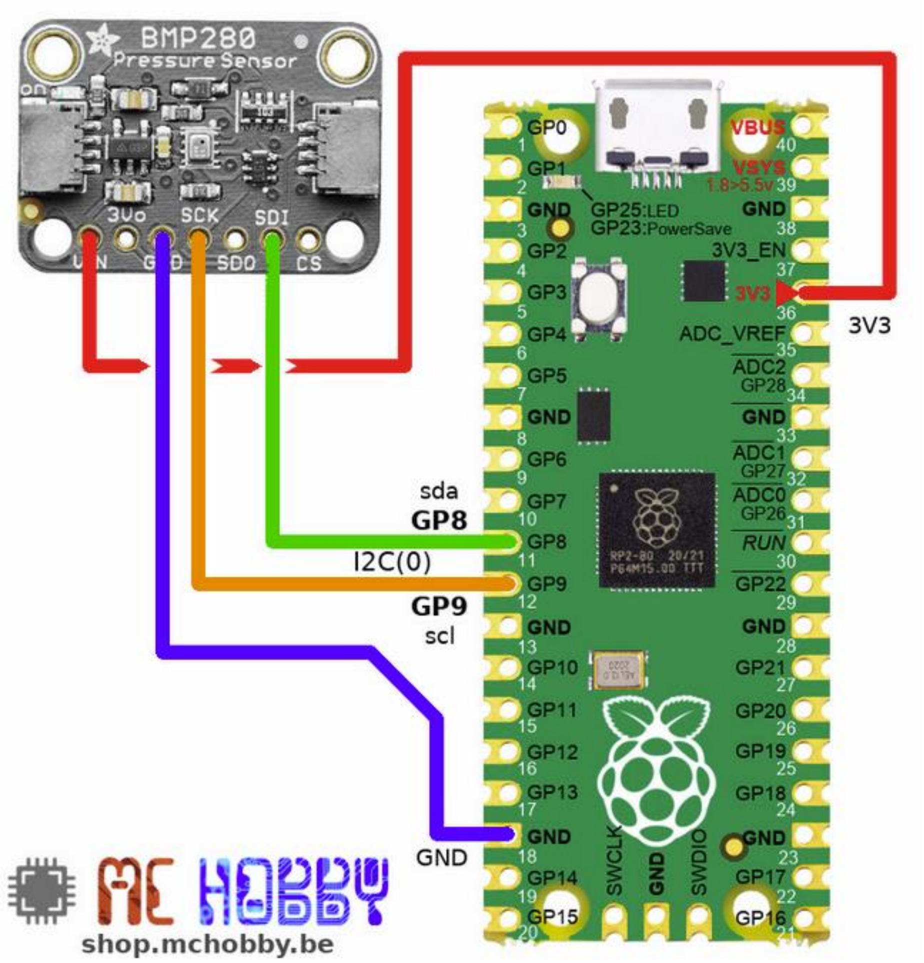
Exercise 2.1: Connecting the BMP280 Pressure/Temperature Sensor using I2C

Before we can read data from the sensor we need to connect it to the Pi. I2C requires us to connect two data cables and the BMP280 sensor also requires VCC (power) and GND (ground) connection, thus four cables in total.

1. Connect the 2-6V input pin on the BMP280 to pin 3V3 on the Pi (3.3 volts 300 mA output) and connect the GND pin on the BMP280 to a GND pin on the Pi.

2. The I2C SCL and SDA pins need connecting to the Pi's I2C pins. The Pi Pico has two physical I2C interfaces that can be configured to use several pairs of pins to fit a PCB design or needs for other interfaces.

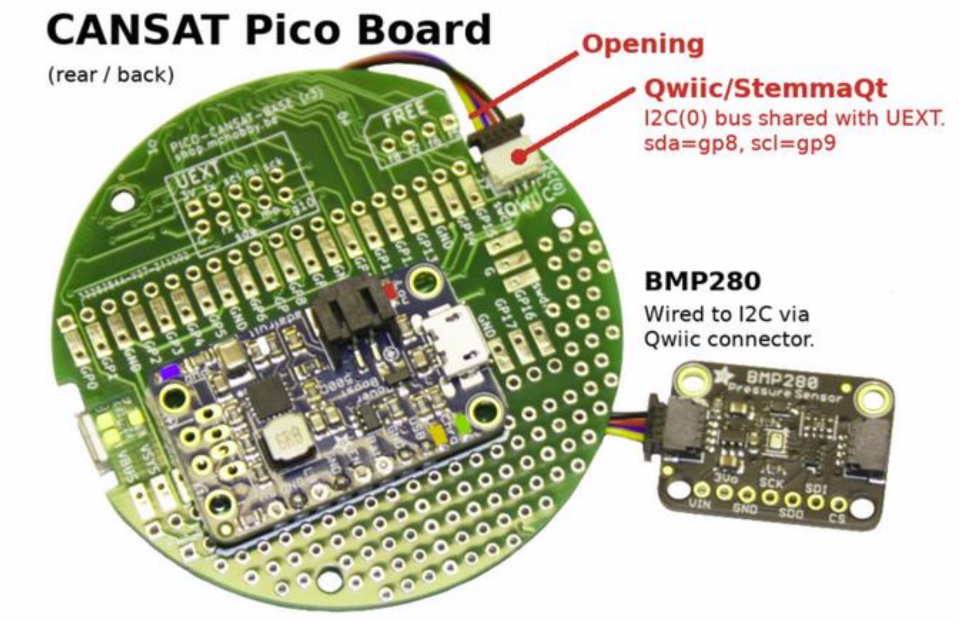
   For now, we will use IC2 0 on pins GP8 and GP9. These are shown in blue in the diagram below:



And so connect the SDA and SCL BMP280 pins to pins GP8 and GP9 on the Pi.

Alternative way to connect the BMP280 sensor and the Pi Pico.

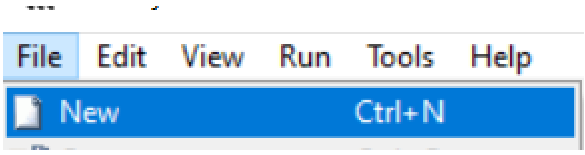
The kit you have received offer the possibility to connect the sensor and the Pi Pico through a cable



Exercise 2.2: Reading the Temperature and Pressure

Now that the BMP280 is connected and set up we can read data from it.

1. Create a new file



2. Save this file under the name you want, terminated by .py

3. We need to use several pre-built libraries to access the BMP280 I2C:

    - *machine :* provides access to the Raspberry Pi Pico pins

      (I2C, SPI, UART)

    - *bmp280:* the BMP280 sensor library provided by the manufacturer

4. Before we read from the sensor, we need set up the I2C interface by telling python which Raspberry Pi pins we would like to use for the interface. We also need to import the library of the bmp280 board.

```
from machine import I2C
from bmp280 import *
```

5. We import the time library as usual to make a pause between two successive readings of the sensor

```
import time
```

6. We now need to create an object that represents the BMP280 sensor using the Adafruit library. We also need to tell the library which I2C interface we wish to use and the I2C address of the sensor, for the sensor in the kits this is I2C 0:

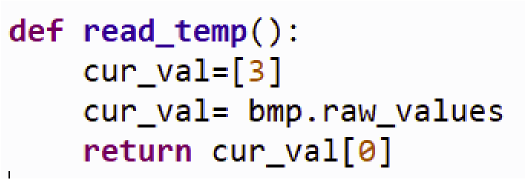
```
  i2c = I2C(0) # sda=GP8, scl=GP9 @ 400 KHz (default)
  bmp = BME280(i2c=i2c, address=BMP280_I2CADDR)
```

You can search on the WEB if you want to have more information about I2C bus on the official MicroPython I2C documentation

7. We can now add a function to read and print one of the values from the BMP280 sensor (temp, atm pressure, humidity)

   the function bmp.raw_values return the 3 values in a row. If we want to display a specific one it’s necessary to create a list which will first store the data returned by the function bmp.raw_values. We select the in raw the value we need

   The function definition to read the temperature (first value in the raw) would be



```
def read_temp():
    cur_val=[3]
    cur_val= bmp.raw_values
    return cur_val[0]
```

   Write a function *read_pressure()* that can read the pressure from the sensor (it will look very similar to the *read_tem()* function), but with one decimal (see example above)

8. You have now written your own BMP280 library. Return to the *code.py* file.

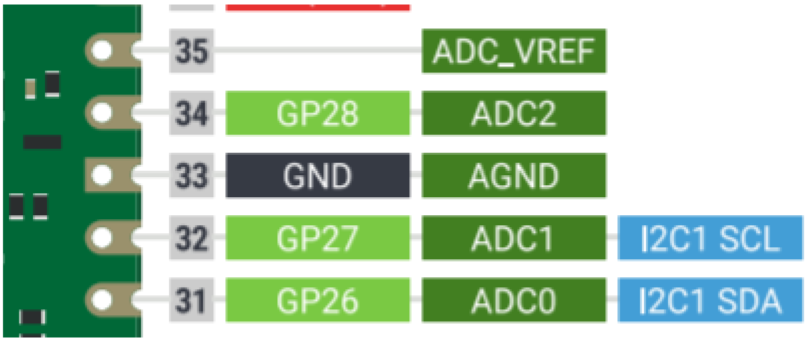
9. Save the code and correct any errors. If there is an error concerning the I2C then check your wiring. Your CanSat should print out the temperature and pressure readings every 1s.

   Add the format string, run the program and observe the difference in output style.

Exercise 2.3 (optional): Interfacing an Analogue Sensor via the ADC

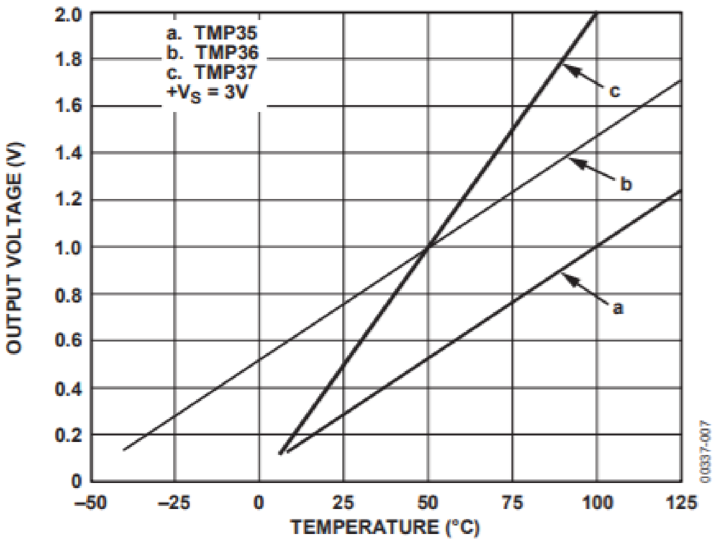
I2C is a digital interface, only two voltage levels are supported (0V and 3.3V) which limits its use as a sensor input when interfacing directly to any analogue electronic sensors you may have or developed or procured. The Raspberry Pi Pico contains three Analogue-to-Digital

Convertors (ADC) that can translate an analogue voltage into a number that the Python program can use. These are located on GP26, GP27 and GP28 as below:



In their default configuration, the Pi ADCs will sample the voltage on the ADC input pin, this must be within the range of 0V to 3.3V. Once sampled, it converts the voltage to a number between 0 and 65,535 with a value of 0 representing 0V and a value of 65,535 representing 3.3V.

The kits include a TMP36 analogue temperature sensor. This sensor outputs a voltage dependant on the ambient temperature the device measures. The output voltage to temperature relationship for a 3V input is as follows:



TMP36 Datasheet Rev H.

Therefore, by connecting the output of the TMP36 to the ADC of the Pi and performing some transformation of the value read, we can measure the temperature using the TMP36.

1. Connect the TMP36 to the Pi as follows:

   a. Pin 1: 3V3

   b. Pin 2: ADC0 (GP26)

   c. Pin 3: GND

(please note that if Pin 1 and Pin 3 are reversed the TMP36 can get very hot quite quickly, so double check before powering up the Pi)

The pins on the TMP36 have the following layout (viewed from the bottom of the device):

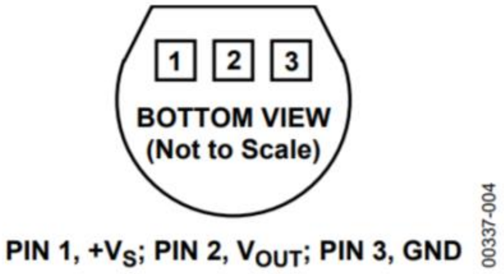
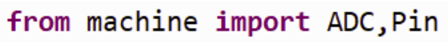


Figure 4. T-3 (TO-92)

Figure 0-1 Source: TMP36 Datasheet Rev H.

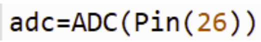
2. We can now write the python code to access the TMP36. Create and save a new file called *tmp36.py*

3. In this new file, import the following required libraries:



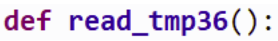
```
from machine import ADC,Pin
```

4. Now setup the ADC associated with pin GP26 with the following line:



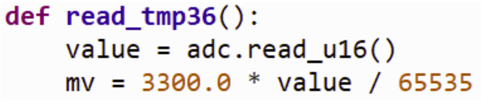
```
adc=ADC(Pin(26))
```

5. To read the sensor via the ADC will we create a function (in the same way as the BMP280 was used). First, create the read_temp36() function.



```
def read_tmp36():
```

6. We then need to read the ADC and convert the value to a voltage by scaling the read value in the range of 0 mV to 3300 mV (0V to 3.3 V):



```
def read_tmp36():
    value = adc.read_u16()
    mv = 3300.0 * value / 65535
```

7. The voltage reading can then be converted to a temperature using the following information from the data sheet:

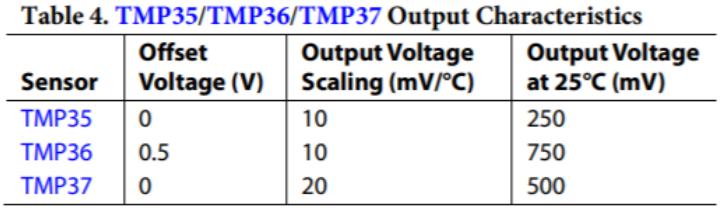
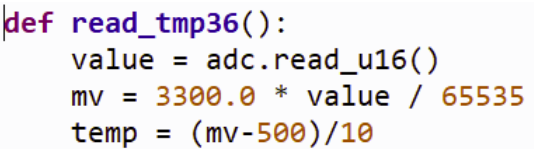


Table 4. TMP35/TMP36/TMP37 Output Characteristics

Sensor	Offset Voltage (V)	Output Voltage Scaling (mV/°C)	Output Voltage at 25°C (mV)
TMP35	0	10	250
TMP36	0.5	10	750
TMP37	0	20	500

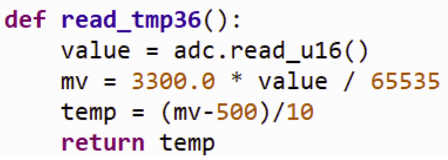
TMP36 Datasheet Rev H.

As we are using the TMP36 our offset (500 mV) needs to be deducted from the voltage read by the ADC and the whole result scaled by 100 (as we are working in V whilst the datasheet is working in mV):



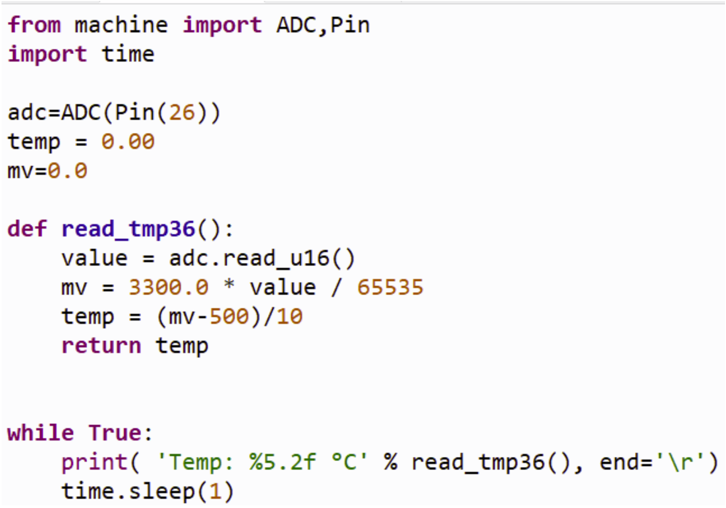
```
def read_tmp36():
    value = adc.read_u16()
    mv = 3300.0 * value / 65535
    temp = (mv-500)/10
```

8. Return the calculated temperature:



```
def read_tmp36():
    value = adc.read_u16()
    mv = 3300.0 * value / 65535
    temp = (mv-500)/10
    return temp
```

9. Then call the read_temperature function and print the result. We have then the resulting complete code



```
from machine import ADC,Pin
import time

adc=ADC(Pin(26))
temp = 0.00
mv=0.0

def read_tmp36():
    value = adc.read_u16()
    mv = 3300.0 * value / 65535
    temp = (mv-500)/10
    return temp


while True:
    print( 'Temp: %5.2f °C' % read_tmp36(), end='\r')
    time.sleep(1)
```

10. Run the code and compare the sensor readings between the TMP36 and the BMP280.

there are several ways to measure the temperature with electrical, actif or passif, sensors. The choice of the sensor depends on the precision as well the range of the temperature wanted, the reactivity needed, the consumption of the energy (the energy in Lipo battery is not infinite and the resulted price is also an important parameter during the choice).

Another very important parameter is the placement of the sensor. To measure the outside temperature during the mission 1 of the CanSat, the sensor has to be placed on the external wall.

Lab 3: SPI Interface and Radio Communication

This lab builds on the sensing and message sending capabilities we have developed in the previous labs by adding wireless capabilities to the CanSat, using the SPI interface. This will fulfil the electrical requirements for the primary mission.

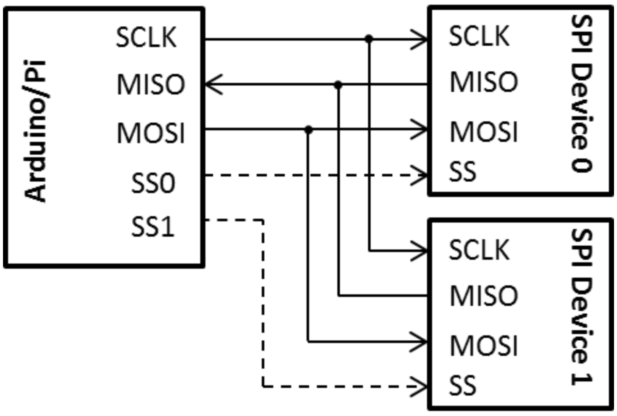
This lab will require two parts to operate, one to send data and one to receive data. Exercise 4.3 builds the CanSat (data transmission) software and Exercise 4.4 builds the Ground Station (data receive) software. We will have a beacon set up at the front of the room that will receive all packets. Alternatively, you can pair with someone else; one taking the CanSat role and the other the Ground Station role.

Serial Peripheral Interface (SPI)

SPI offers an interface with more powerful capabilities than I2C at the cost of more wiring required. As with I2C it also supports bi-directional communication with several devices but offers a much higher data throughput. This makes it suitable for communicating with the most complex devices that you might connect to the CanSat. The interface consists of at least four pins:

SCLK: *Serial Clock*. A stream of 0-1s that the data is aligned to. The SPI clock rate is related to the speed of this stream, you can slow this down if having data integrity issues.
MISO: *Master Input / Slave Output.* The data from the peripheral device to the Pi.
MOSI: *Master Output / Slave Input.* The data from the Pi to the peripheral device.
SS0/CE0: *Slave Select / Chip Enable.* Enables a peripheral device and means that the device can output to the MISO pin. One SS/CE pin is needed for each peripheral device.

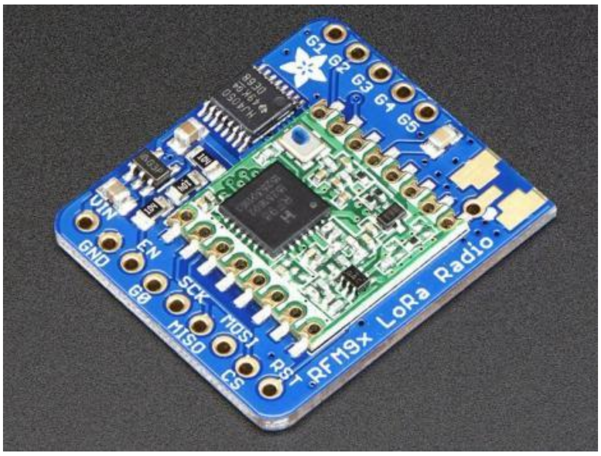
To use SPI you don't need to be too concerned about the function of these pins as the device's software library will take care of most of the low-level SPI code for you. However it is good to be aware of their function when cascading multiple SPI devices together, for example to connect two devices you will need two SS/CE pins:



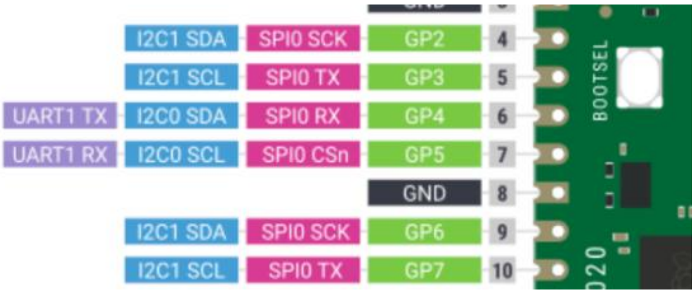
Typical SPI CanSat Uses:
Cameras, Storage cards (e.g. SD cards), GPS modules, WiFi Modems

Exercise 3.1: Connecting to the RFM69 LORA Radio Module using SPI

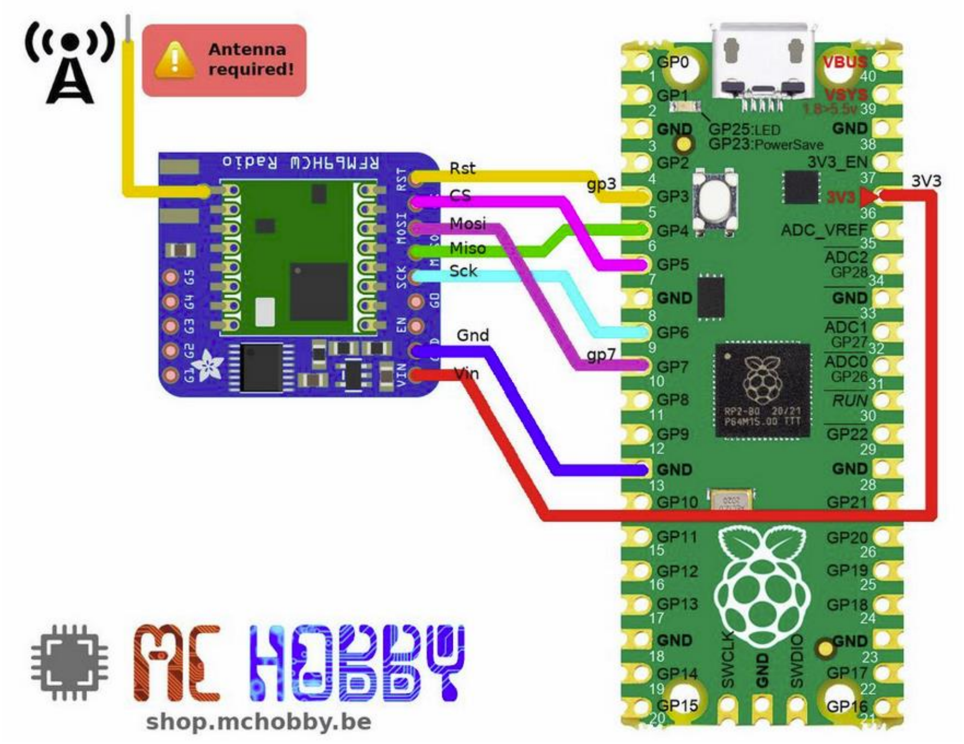
The RFM69 LORA module is a long range (upto 2km line-of-sight), low throughput, radio module and connects to the RaspberryPi via an SPI connection. The SPI signals are present on the RFM96x as SCK (SCLK), MISO and MOSI.



https://learn.adafruit.com/adafruit-rfm69hcw-and-rfm96-rfm95-rfm98-lora-packet-padio-breakouts/pinouts



The Raspberry Pi Pico has two SPI interfaces and, as with the UART and I2C interfaces, it can be setup to use a variety of pins for the interface. For this lab we will use the GP2 to GP7 pins for the radio.



The wiring between the Pi Pico and the RFM90 module is the same for the the base station or the CanSat station.

For the base station it is easier to connect the chips on the breadboard.

For the CanSat station it its recommended to first connect the

1. Connect the power signals on the RFM69. You will need two cables to connect the VIN and GND pins on the RFM69 to the 3V3 and GND pins on the Pi. This module can cope with both 3.3V and 5V signals, but as the Raspberry Pi's logic pins are 3.3V we use that voltage for the RFM69. Pin 36 provides VCC (or it can be chained from the BMP280’s Vin pin) and there are several GND pins to use.

2. Connect the three SPI signals (SCK, MISO, MOSI) from the RFM69 module to the Raspberry Pi. GP2 will be used for SCK, GP3 for MOSI (Master-Out, Slave-In, SPI0-TX on the Pi) and GP4 for MISO (Master-In, Slave-Out, SPI0-RX on the Pi).

3. The RFM69 needs the SPI chip select pin. Connect the CS pin to a GPIO pin so that we can set this to zero to reset the RFM69, pin GP6 is used in this example.

4. The RFM69 needs to be reset on start up. Connect the RST pin to a GPIO pin so that we can set this to zero to reset the RFM69, pin GP7 is used in this example.

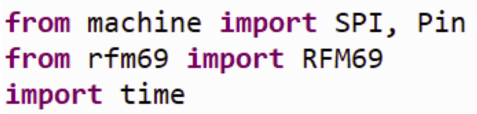
   At this point you should have 7 wires (2 power, 3 SPI, CS, reset) connected (and the BMP280 wires if you have left those connected).

Exercise 3.2: Setting up the RFM69 Radio

Now that the hardware is connected, we configure the software side of the radio module.

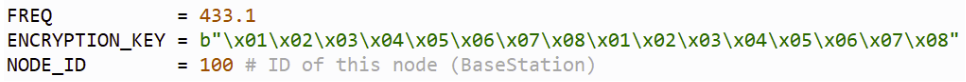
1. Create a new file and save it as *radio.py*

2. First we need to add the required libraries. As with the I2C sensor, we need to add the board and bus io libraries to access the SPI interface. We also need the digitalio library for the CS and reset pins and finally we also need the RFM69 radio library.



```
from machine import SPI, Pin
from rfm69 import RFM69
import time
```

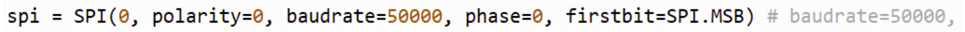
3. We declare constants for the frequency of the radio signal, the encryption key and the node ID which is the identification of the station



```
FREQ           = 433.1
ENCRYPTION_KEY = b"\x01\x02\x03\x04\x05\x06\x07\x08\x01\x02\x03\x04\x05\x06\x07\x08"
NODE_ID        = 100 # ID of this node (BaseStation)
```

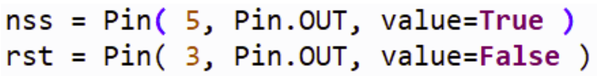
Encryption key has to be unique for each couple base station / can station. The node ID have to be unique for each station. These changes are necessary in order to avoid interference with the other projects.

4. Now we setup the SPI interface so that we can communicate to the RFM69. We map the SPI signals to the pin numbers based on how they are connected.



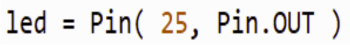
```
spi = SPI(0, polarity=0, baudrate=50000, phase=0, firstbit=SPI.MSB) # baudrate=50000,
```

5. We need to also set up the CS or NSS and reset pins as GPIO digital pins:



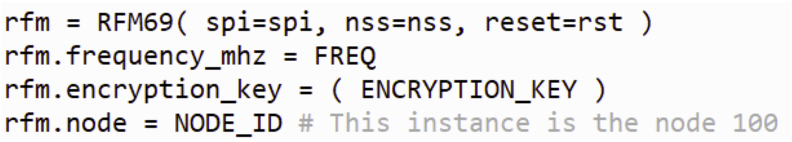
```
nss = Pin( 5, Pin.OUT, value=True )
rst = Pin( 3, Pin.OUT, value=False )
```

6. We use the build-in led to give indication of the received or transmitted nformation



```
led = Pin( 25, Pin.OUT )
```

7. We can now start up the radio. To do this we can call the RFM69 library functions, this will give us an object that we can then use to represent the radio:



```
rfm = RFM69( spi=spi, nss=nss, reset=rst )
rfm.frequency_mhz = FREQ
rfm.encryption_key = ( ENCRYPTION_KEY )
rfm.node = NODE_ID # This instance is the node 100
```

As you can see, the RFM69() function takes several parameters:

- The SPI interface to use
- The CS (cheap select) or NSS (slave select) and reset pins to use
- Operating frequency: Several different RFM69 are available that operate at different frequencies, 433MHz is recommend to use in the UK, Belgium and Luxemburg as it is part of a free ISM radio frequency allocation. Therefore the operating frequency is set to 433.0MHz
- The *RFM69* object that this function returns is what we shall use for accessing the radio for other parts of the lab.

These lines of the code *radio.py* are common for the *cansat.py* program (which read and transmit the measured values) as for the *basestation.py* (which receives the transmitted values from the CanSat). There are some differences like the NODE_ID (to change with constant value) and the radiolink quality that we will read in the *basestation.py*.

Exercise 3.3: Radio Data Transmission (CanSat)

We have coded programs to read parameters like the temp, pressure from the bmp280 or the TMP36, we have written the code for the transmission/reception. Now we have to assemble the various codes in one code cansat.py.

Here is the link for the code for the Cansat

https://github.com/mchobby/cansat-belgium-micropython/blob/main/mission1/cansat.py

Proceed as described before for the bmp280.py and rfm69.py

Look at this code you will see that the constants NODE_ID and BASESTATION_ID have been defined and declared. In this the way CanSat communicate to a particular BaseStation. So please make a change for each ID in order to avoid communication with other projects.

There is also another way to avoid the distribution of information to other BaseStation ...

Exercise 3.4: Radio Data Receive (BaseStation)

As with the transmit, the radio setup of Exercise 3.2 is enough to allow us to start receiving data. Each message sent over the radio is dubbed a *packet* and the receiver can receive one packet at a time.

Here you will find the code to implement in the BaseStation.

https://github.com/mchobby/cansat-belgium-micropython/blob/main/mission1/basestation.py

Proceed as before. And look attentively to the code.

As for the cansat.py program you will observe that there are a lot of comments (each comment begins with a #). This is very important to comment your code in order to let other developers to your code and also to understand later on to understand your own code.

The code uses another function of the object rfm69, that it is not used in the cansat.py program : rfm.last_rssi This function gives the radio link quality measurement.

This parameter will indicate if the antennas are well designed. Use this parameter during the construction of your antenna on the ground before the launch of your satellite !!

Run the code and check that you receive packets from either the beacon at the front of the room or from your neighbour undertaking the transmission exercise.

Recording the data from the space !

The program your developed has read the captors to observe the temperature the atmospheric pressure. But if we want to analyse the data we need to store them in a file with the values and a time stamp so that we can afterwards make graphics with a tool like excel.

For that we have to create the file (with a file name and the location of this file) and we have to close it when the flight is finished. The data can be stored on the imbedded board (satellite station) or on the base station or both !. It's recommended to store on both sides

If we choose to record the data on the base station during the flight, we should open a file. Let's choose a name that we will remember after a while. For instance CanSatRec.txt. This indicates the data will be text.

Therefore, in the receive.py program we add a line at the beginning of the program (after the import lines and the base settings

CSR=open ("CanSatRec.txt","a"). "a" stands for append so that each received value will not erase the previous one.

And each time the base station receive a new data we will store it in the file, choosing the string type before to record

CSR.fwrite(packet packet_txt)

And at the end we close the file. CSR.close()