CanSat RaspberryPi 2023 Workbook

The following four labs have been designed to introduce you to some of the electronics and programming skills that will be required to undertake the CanSat competition. Whilst these will be delivered as part of the CPD workshops, these lab scripts have been written in a standalone fashion to allow you to finish or repeat any of the exercises outside of the workshop.

The Teaching Kit

The teaching 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 Circuit Python. The kit we use in the labs contains the following items:

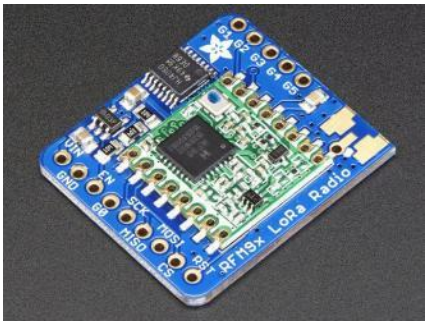
Raspberry Pi Pico

The Raspberry Pi Pico combines a small microcontroller with several hardware interfaces (UART, SPI, I2C) that have a large amount of configurability. USB programming support and a large memory make this an easy microcontroller to work with and is ideal for getting started with CanSat

In these labs we install CircuitPython on the Pi Pico to allow the device to be programmed using the Python programming language. For better performance, lower memory usage and smaller code size it may be worth considering using the Ardunio platform as an alternative to CircuitPython

https://uk.farnell.com/raspberry-pi/raspberry-pi-pico

Adafruit RFM96W LORA Transceiver Radio Breakout - 433 MHz



The RFM96W is our communication device. It connects to the Raspberry Pi and allows messages to be sent via radio with a range of around 2KM line-of-sight. It comes in several available frequencies but 433MHz is recommended of UK use.

It is provided without an antenna but either a 173mm wire can be soldered to the ANT pin or a SMA connector can be attached for connection to a larger gain antenna such as YAGII.

https://www.adafruit.com/product/3072

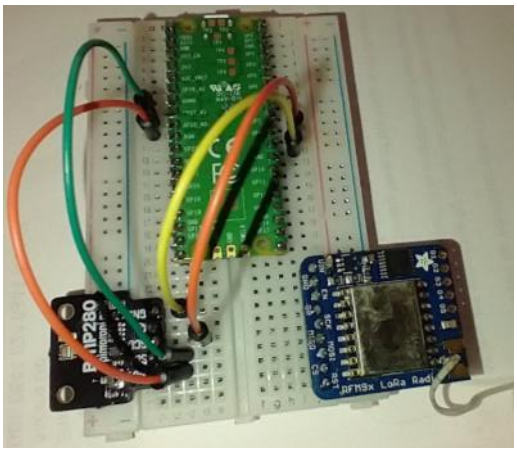
BMP280 Barometric Pressure/Temperature/Altitude Sensor

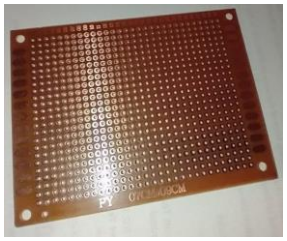
The BMP280 allows temperature and pressure to be read from a single sensor via the I2C interface.

https://shop.pimoroni.com/products/bmp280-breakout-temperature-pressure-altitude-sensor

Other Items in the Kit

Breadboard - for protoyping circuits





Copper-clad protoboard - for prototypes or assembling the CanSat. Components can be soldered to the board and connected to each other via solder or wiring.

2.54mm Pin Headers - For soldering to the boards or to the protoboard for connecting to cables and other boards.

Cables for connecting the boards together

Lab 0: Setting Up the Programming Software

A new Raspberry Pi Pico will need CircuitPython and the required sensor python libraries setting up before these labs can be followed.

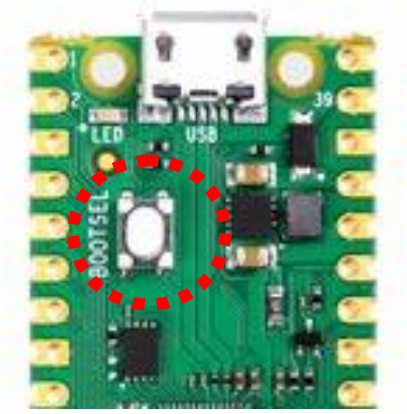
Install CircuitPython on the Pi Pico

CircuitPython is a Python environment for controlling small computer systems such as the Pi Pico. It allows such boards to be programmed in Python by uploading a main python file (*code.py or main.py)* that is executed when the Pi Pico is switched on. It contains functions for direct low-level control of the board's pins and hardware interfaces.

Follow the following steps to install CircuitPython on the Pi Pico.

1. Download the CircuitPython environment for the Pi Pico from https://circuitpython.org/board/raspberry_pi_pico/ or from the memory stick at the training session. This should be a ".uf2" file.

2. The Pi Pico needs to be started into a different mode to install the CircuitPython environment. To do this, hold down the BOOTSEL switch and plug the USB cable into the Pi and your PC.



3. The Pi should mount as an USB drive labelled *"RPI-RP2"*, at which point release the BOOTSEL button.

4. Copy the .uf2 file onto the Pi USB drive. The USB drive will unmount itself and then re-mount as a new USB drive labelled "CIRCUITPY"

5. If you are using Windows 7 or 8, you will also need to download and install the "Adafruit drivers" package so that Windows can communicate with the Pi Micro. https://github.com/adafruit/Adafruit_Windows_Drivers/releases

6. If your Raspberry Pi Pico becomes unresponsive (e.g. doesn't show up when plugged into a PC) then you can try repeating these steps to recover it. Should re-copying the CircuitPython .uf2 not save your board then as a last resort try coping the

"flash_nuke.uf2" file at Step 4, available here https://learn.adafruit.com/getting-started-with-raspberry-pi-pico-circuitpython/circuitpython . This .uf2 file will wipe the PI and restore all settings to default values. You will need to rerun these steps after "nuking" your Pi Pico!

Install BMP280, RFM9x and Adafruit Boards libraries

CircuitPython 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 CircuitPython 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.

Libraries are installed into CircuitPython by copying across the *".mpy"* file associated with the library to the *"lib"* folder found on the USB drive of the Pi Pico.

For the CanSat primary mission we need to install the BMP280 sensor library and the RFM9x radio library. These are available from the Adafruit CircuitPython library collection which is available from the following location:
https://github.com/adafruit/Adafruit_CircuitPython_Bundle/releases

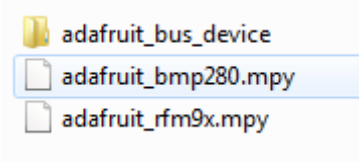
1. Download and extract the latest version of adafruit-circuitpython-bundle-7.x-mpy.zip or from the USB stick in the training session.

2. Find *adafruit_bmp280.mpy* within the *lib* folder of the adafruit library collection. Copy this file over to the Pi Pico USB drive (CIRCUITPY).

3. Find *adafruit_rfm9x.mpy* within the *lib* folder of the adafruit library collection. Copy this file over to the Pi Pico USB drive (CIRCUITPY).

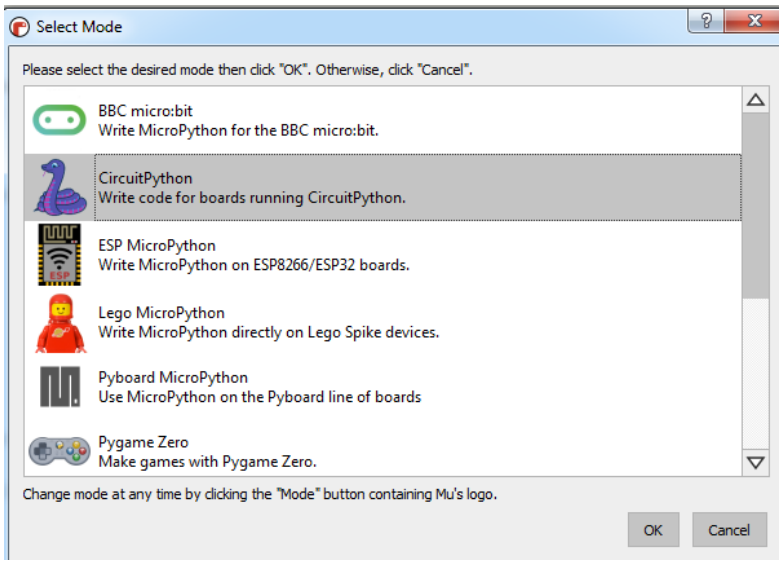
4. Both of these libraries depend on another library to work, this is the *adafruit_bus_device* library. This library is a collection of *.mpy* files so look for a folder with that name in the Adafruit library collection. Copy the whole folder over to the *lib* folder of the Pi Pico.

5. The contents of your *lib* folder on your Pi Pico should now look as follows:



Mu Editor with CircuitPython

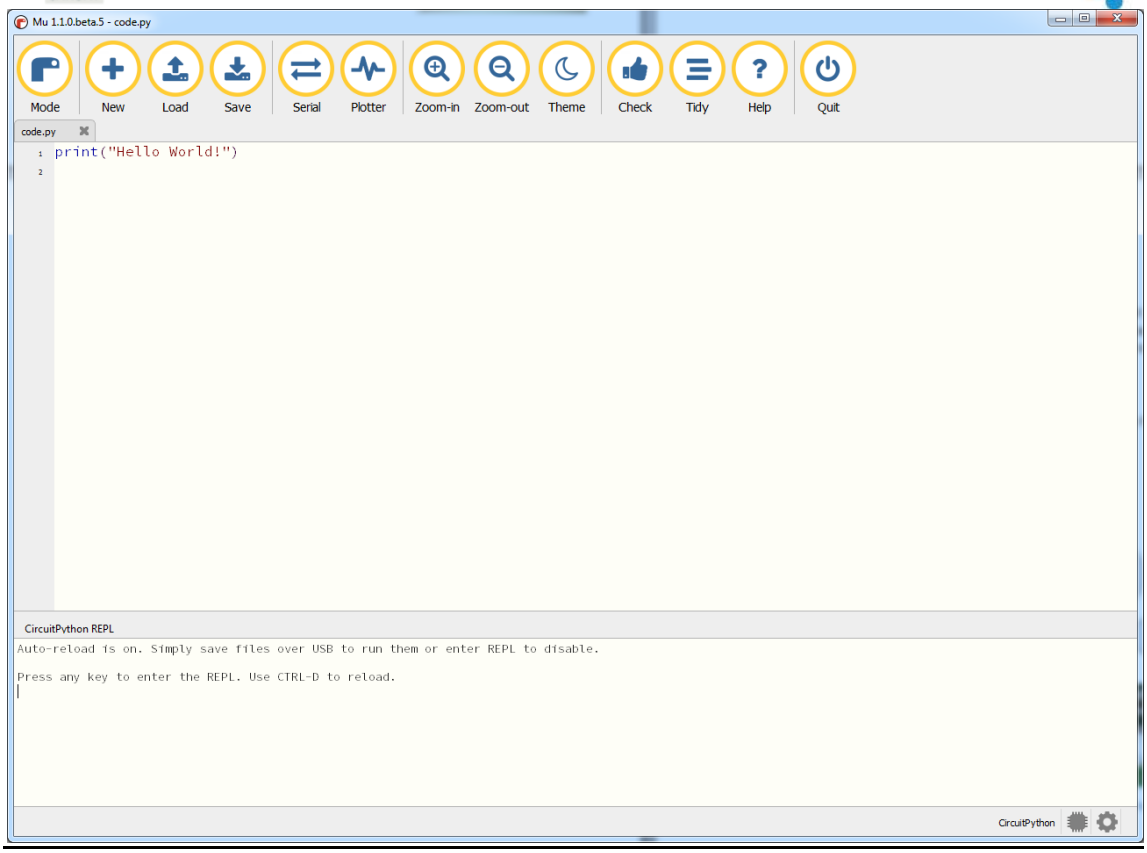
There are several programs available to develop code for CircuitPython. Mu Editor https://codewith.mu/ is simple to use and has built-in support for CircuitPython. When starting Mu you will be asked for what mode Mu should operate in. Chose CircuitPython at this screen:



The Mu Environment is fairly self-explanatory. When used in CircuitPython mode, pressing the Save button or ctrl-S will reset the Pi Pico and so run the code you have just edited. The serial window will show you the output of any *print()* commands and allow you to interact with the Python REPL (as we see in Lab 1).

Clicking on the Serial Window and pressing a key will take you to the REPL.

To quit the REPL press ctrl-D.



Lab 1: Analogue Sensing and Soldering

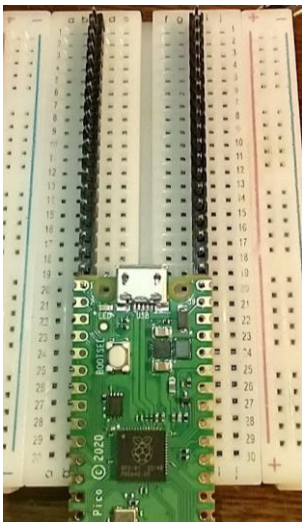
The first lab will cover running a basic Python3 program that tests that CircuitPython and the Pi Pico are up and running correctly. It also contains some soldering to build the boards in your kit. Soldered connections are one of the most reliable ways of connecting parts of the CanSat electrical design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

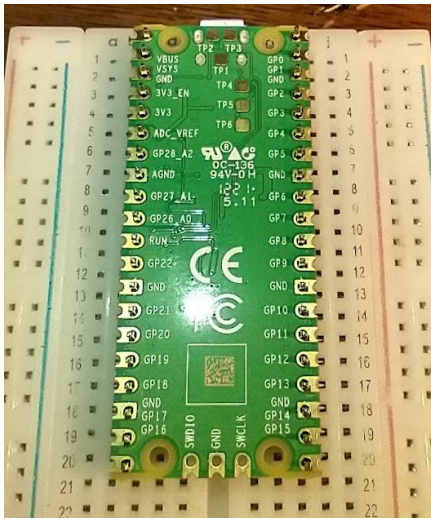
Exercise 1.1: Soldering the CanSat Kits

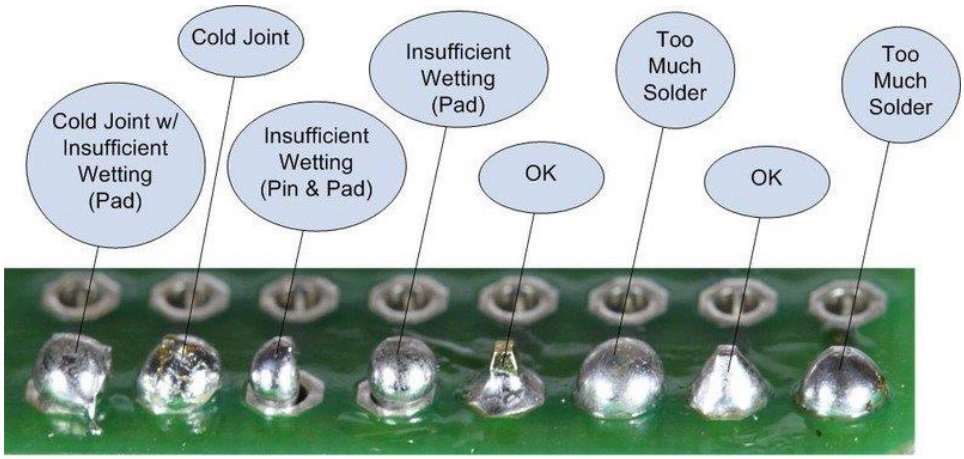
The RFM96W, BMP280 and Raspberry Pi Pico boards will need header pins soldering to them so that they can be used with the breadboard and jumper wires.

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.





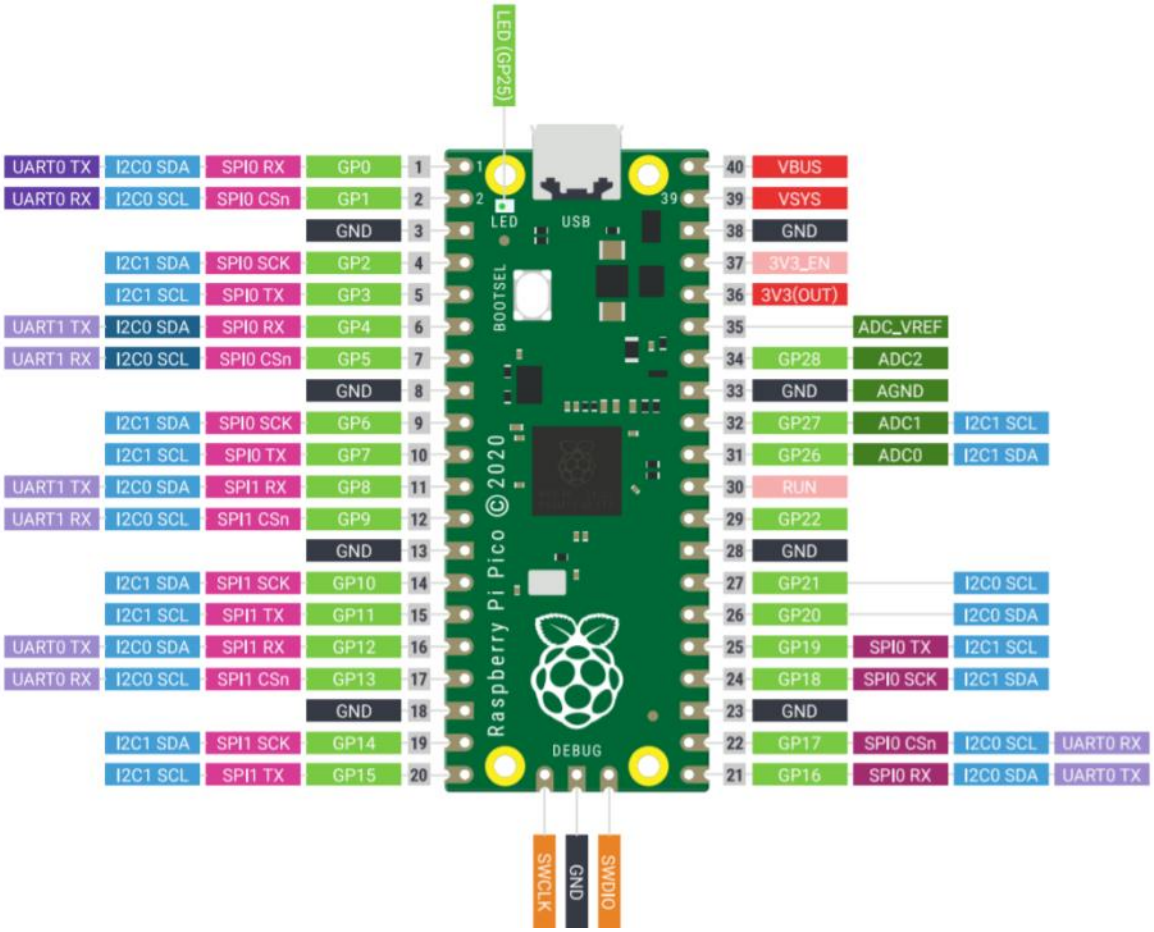


https://learn.adafruit.com/adafruit-guide-excellent-soldering/common-problems

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 as we shall see in this lab. 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

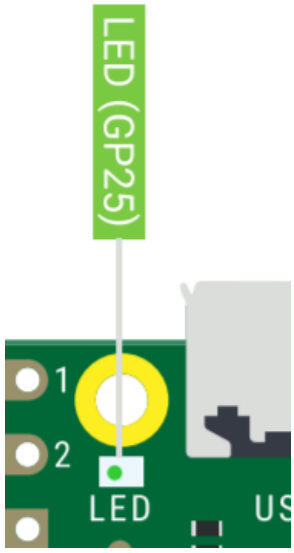


https://datasheets.raspberrypi.org/pico/Pico-R3-A4-Pinout.pdf

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

Exercise 1.2: Controlling the LED from REPL Using GPIO

The Pi Pico has a built in LED on GP25:



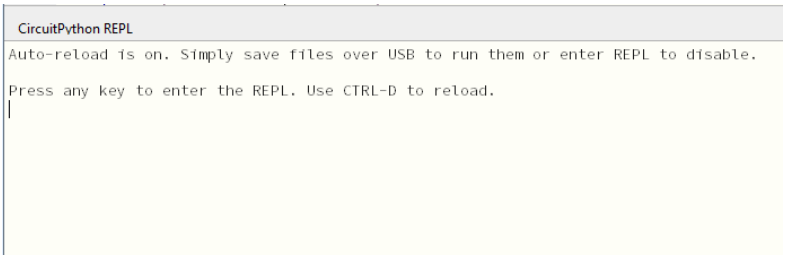
As this is a GPIO pin, we can control it from the CircuitPython software. To test this will we use the *Read-Evaluate-Print-Loop (REPL)* functionality of Python that allows us to write basic code without saving it to a file. The code has to be entered one line at a time, which can be tedious but is useful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

1. Connect the Raspberry Pi Pico to the laptop.

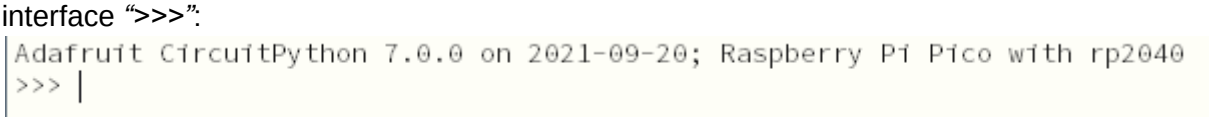
2. Open the Mu editor.

3. Open the “Serial” window by clicking the button on the toolbar:

4. The following window should appear:



5. Click on this window and press enter. You should be prompted with the Python REPL interface *“>>>”*:



6. 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 two Python libraries. The first provides the GPIO functions, type the following into REPL and press enter:

```
>>> import digitalio
```

7. The second library contains the pin names for the Raspberry Pi Pico, type the following into REPL:

```
>>> import board
```

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

```
>>> led = digitalio.DigitalInOut(board.GP25)
```

This creates an object called *led* that we can use to interact with the LED pin

9. Then set the GPIO to output (GPIOs can be either input or output pins)

```
led.direction = digitalio.Direction.OUTPUT
```

10. Now we can control the LED from Python. The following line should turn on the LED:

```
>>> led.value = True
```

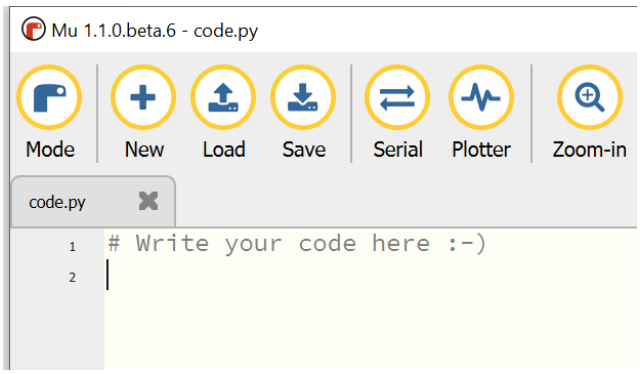
11. Now write a command to switch the LED off.

Exercise 1.3: Controlling the LED from Software Using GPIO

The CircuitPython REPL is handy for testing small amounts of code, but for the CanSat application the code will need to be written into a file. This file is saved to the Pi Pico and will be automatically run when your CanSat is powered up.

1. Close the REPL connection by pressing Ctrl-D in the REPL window. This is a useful command should you enter REPL mode by accident.

2. The main editor window in Mu should have a file called *code.py* already open. This is the main file that CircuitPython will run on start-up of the Pi.



3. In this window, retype the above code:

```
import digitalio
import board

led = digitalio.DigitalInOut(board.LED)
led.direction = digitalio.Direction.OUTPUT
led.value = True
```

4. Click Save (or repress Ctrl-S) and check the Serial window below. You should see some text saying that your code has been saved onto the Pi and is running. Check that the LED has turned back on. If so, then this shows you can upload and run code on your Pi.

5. We can make the LED blink by adding a time delay between switching the LED on and off and then making this behaviour loop. Add the following code to the end of your file:

```
while True:
    led.value = False
    time.sleep(0.5)
    led.value = True
    time.sleep(0.5)
```

   The *sleep()* function will delay the program by the number of seconds in the argument (in this case, half a second). The LED is turned off, the program sleeps for half a second, then the LED is turned back on and the program sleeps again. At this point the program returns to the top of the *while* loop checks that True == True (it always is!) and runs the loop again.

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

6. Run the code. You will see some errors regarding the *sleep()* function. The sleep function is part of the “time” library, as seen by the *“time.”* before the function is called. Therefore, add some code to import the *time* library in the same way as the board and digitalio libraries were added.

7. The code should now run.

8. 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.value = not led.value
    time.sleep(1)
```

9. 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: Interfacing an Analogue Sensor via the ADC

GPIO 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 pin ADC_VREF is the reference voltage that the ADC compares the input voltage to, likewise AGND is the ground pin of the ADC circuitry inside the PI. For cleaner analogue readings it is important that these pins are used as part of the measurement.

1. Create and save a new file called *mf52.py*

2. We shall first setup the ADC. Import the required libraries to access the ADC pins:

```
import board
import analogio
```

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

```
adc = analogio.AnalogIn(board.GP26)
```

4. ***Read the ADC with nothing connected***
   To read the sensor via the ADC will we create a function (in the same way as the BMP280 was used). First, create a read_temperature() function. Then add the following lines to read the ADC and print the value recorded:

```
adc_raw = adc.value
print(f"Raw ADC value: {adc_raw}")
```

5. Now return to the *code.py* file

6. Import your *mf52.py* library by adding the following to the top of the file:

```
import mf52
```

7. Add the function call to *mf52.read_temperature()* within your while loop to call the function once a second.

8. Run your code, note the values that are printed out with nothing connected to the ADC. Is this what you would expect?

9. Modify read_temperature() to also print the ADC voltage. This can be done by multiplying the adc_raw value by the ADC_VREF value (3.3 V) and dividing by the maximum raw reading of the ADC (65,536).

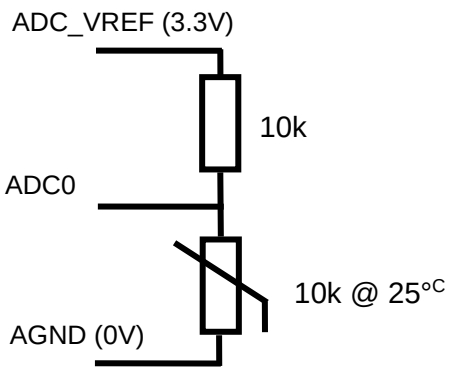
10. Check that your displayed voltages are always between 0V and 3.3V

11. ***Read the ADC with 10k resistor between ADC0 and 3.3V***
    Using the breadboard, plug the 10k resistor between the ADC0 and 3.3V pins of the Pi. Rerun your code and check the voltages measured.

12. ***Read the ADC with 10k resistor between ADC0 and ADC_VREF***
    Move the 10k resistor so that it is now between ADC0 and ADC_VEF. Note the voltages measured and the difference with 3.3V. Hold your finger on the resistor to heat it up slightly and see if there is a change in voltage.

13. ***Read the ADC with the thermistor also connected between ADC0 and AGND***
    Connect the thermistor between ADC0 and AGND. You have made the following *potential divider* circuit:

$$Vout = \frac{R2}{R1 + R2} . Vin$$

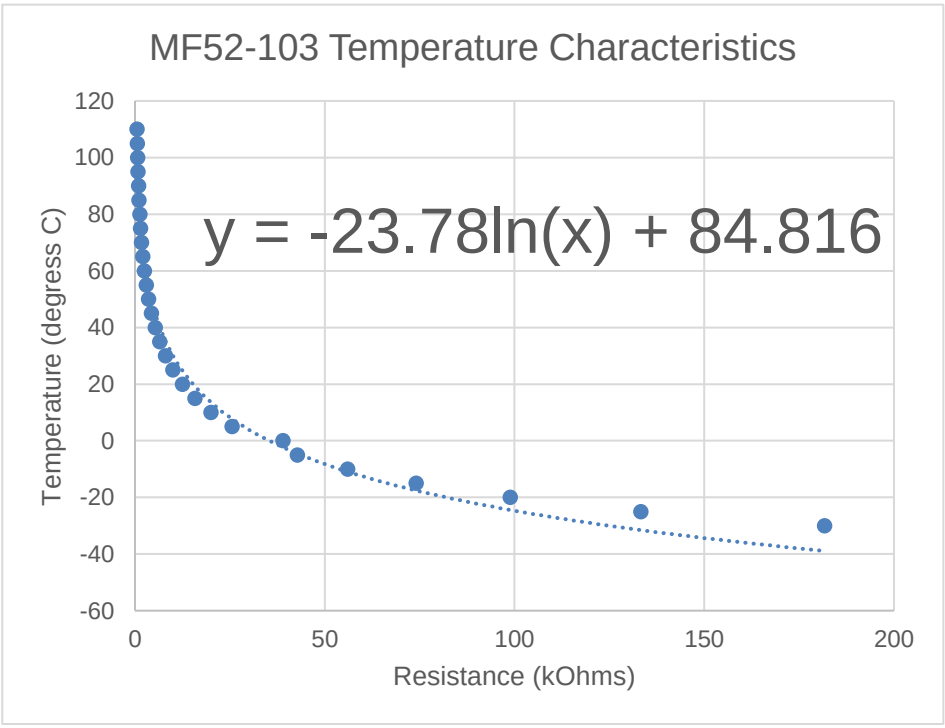


When the resistance in the top half is equal to the resistance in the bottom half, the divider at point ADC0 will be half the input voltage (3.3 V). Check that this is represented in your ADC voltage readings.

The thermistor is a MF52-103, which is a Negative Temperature Coefficient (NTC) thermistor. Thermistors of this type will decrease their resistance with increasing temperature. Decreasing resistance in the thermistor (R2) will decrease the voltage measured at ADC0.

14. ***Converting the ADC voltage to temperature***
    NTC thermistors are non-linear and there are several standard methods (with varying precision) for converting the measured ADC value to temperature. The manufacture's datasheet will have suggestions and formulae. For our thermistor, the following table is provided characterising the sensor resistance at various temperatures. A basic regression line gives the following approximation:

T(℃) \ R25 / B / Rt	10 KΩ
	3950
-30	181.70
-25	133.30
-20	98.88
-15	74.10
-10	56.06
-5	42.80
0	98.96
5	25.58
10	20.00
15	15.76
20	12.51
25	10.00
30	8.048
35	6.518
40	5.312
45	4.354
50	3.588
55	2.974
60	2.476
65	2.072
70	1.743
75	1.473
80	1.250
85	1.065
90	0.911
95	0.7824
100	0.6744
105	0.5836
110	0.5066



MF52-103 Resistance Table https://www.gotronic.fr/pj2-mf52type-1554.pdf

First, we need to convert the ADC value to resistance. This can be done by rearranging the potential divider equation for R2:

```
therm_r = (adc_v * 10000) / (3.3 - adc_v)  #10000 is R1, 3.3 is ADC_VREF
```

Now use this value with the regression formula above to approximate the ADC temperature. You will need to diving the therm_r value by 1000 as the regression formula is in kOhms.

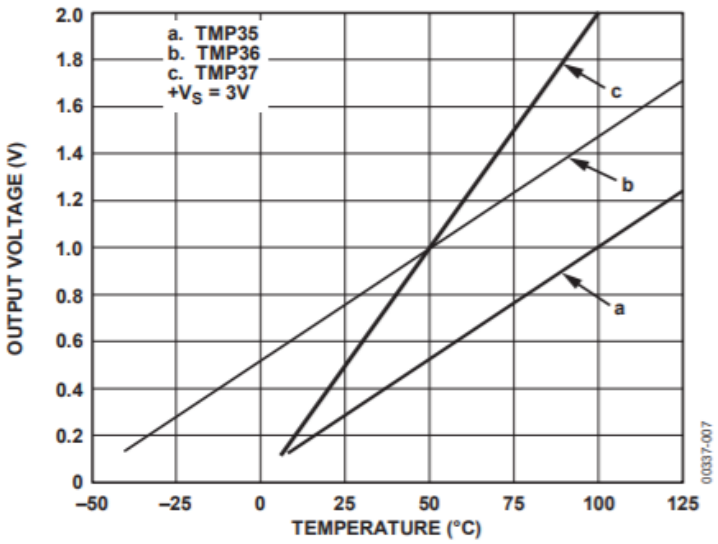
`math.log()` will give you the *ln()* function (remember to import the math library).

Print out the temperature calculated and return the temperature value to end the function:

```
print(f"{temperature}")
return temperature
```

Exercise 1.5 (Optional): Interfacing to the TMP36 analogue temperature sensor

The TMP36 is a different kind of 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: ADV_VREF
    b. Pin 2: ADC1 (GP27)
    c. Pin 3: AGND
    (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):

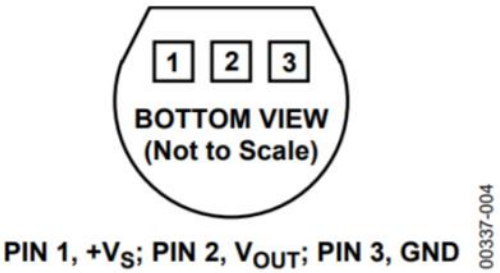


Figure 4. T-3 (TO-92)

TMP36 Datasheet Rev H.

1. Create and save a new file called *tmp36.py*

2. Set up ADC1 in the same way as you set up ADC0. Make a new function called read_temperature_tmp36()

3. In this function read ADC1 and convert the value to a voltage by scaling the read value in the range of 0V to 3.3V:

4. 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 (0.5V) 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):

5. Return the calculated temperature (and add a print for debug):

```
print("raw = {:5d} volts = {:5.2f} temp = {:5.2f}".format(adc_raw, adc_voltage, temperature))
return temperature
```

6. We can now call this function from the while loop within the main *code.py* file.

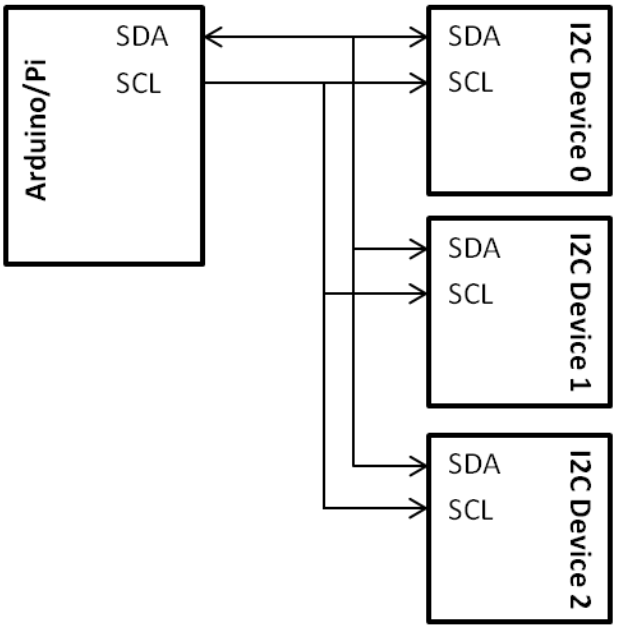
7. Run the code and compare the sensor readings between the TMP36 and the thermistor.

Lab 2: Reading from Digital Sensors (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^2C.

I^2C 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 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 I2C1 on pins GP14 and GP15. These are shown in blue in the diagram below:

Figure 0-1 From https://datasheets.raspberrypi.org/pico/Pico-R3-A4-Pinout.pdf

   And so connect the SDA and SCL BMP280 pins to pins GP14 and GP15 on the Pi.

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 and save a new file called *bmp280.py*
2. We need to use several pre-built libraries to access the BMP I2C:
   - *board:* provides access to the Raspberry Pi Pico pins
   - *busio:* provides access to the Raspberry Pi Pico interfaces (I2C, SPI, UART)
   - *adafruit_bmp280:* the BMP280 sensor library provided by the manufacturer

   Add code to import these files at the start of *bmp280.py*:

```
import board
import busio
import adafruit_bmp280
```

3. 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. Add the following line under the import statements:

```
i2c = busio.I2C(scl = board.GP15, sda = board.GP14)
```

4. We now need to create an object that represents the BMP280 sensor using the Adafruit library. We 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 0x76:

```
bmp280_sensor = adafruit_bmp280.Adafruit_BMP280_I2C(i2c, address=0x76)
```

   This gives us an object, `bmp280_sensor` that we can use to access the I2C BMP280 sensor without having to know any of the low-level I2C transactions.

5. We can now add a function to read the temperature from the sensor. The code

```
bmp280_sensor.temperature
```

   will read the temperature from the sensor. Add the following code to the end of your *bmp280.py* file to create a function that will read the sensor temperature (note the spacing before the return statement!):

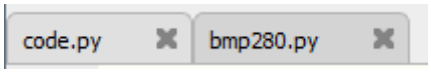
```
def read_temperature():
    return bmp280_sensor.temperature
```

6. The pressure can be read from the sensor with the following code:

```
bmp280_sensor.pressure
```

   Write a function *read_pressure()* that can read the pressure from the sensor (it will look very similar to the *read_temperature()* function).

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



8. We will add some code that reads the temperature and pressure. After the line that toggles the LED within the while True loop, add the following lines to read the temperature and then print it out:

```
cansat_temperature = bmp280.read_temperature()
print(cansat_temperature)
```

9. Add some code so that the sensor also measures and prints out the pressure.

10. 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.

11. It is also possible to print out a message that can combine the values with text and set the precision of the decimal point. This uses the *format string* style of text output:

```
print("T: {:.3f} P: {:.2f}".format(cansat_temperature, cansat_pressure))
```

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

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.

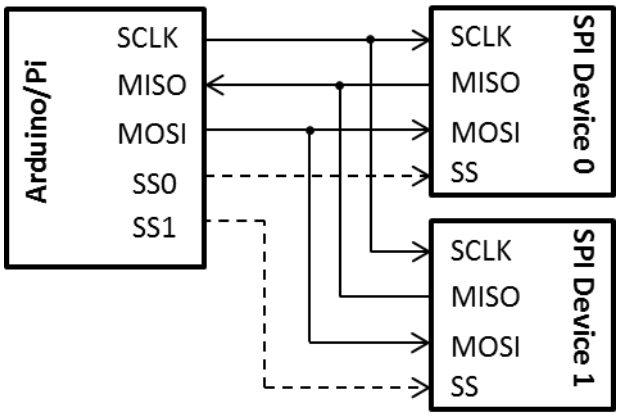
This lab will require two CanSats to operate, one to send data and one to receive data. Exercise 4.3 builds the CanSat (data transmission) software and Exercise 4.4 builds the Ground Station (data receive) software. We will have a beacon set up at the front of the room that will receive all packets. 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.

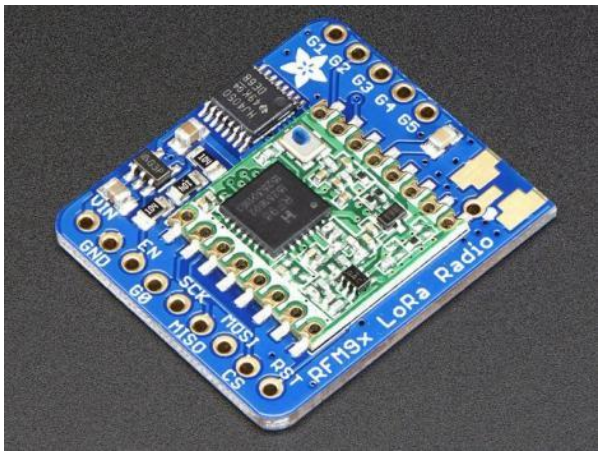
To use SPI you don't need to be too concerned about the function of these pins as the device's software library will take care of most of the low-level SPI code for you. However it is good to be aware of their function when cascading multiple SPI devices together, for example to connect two devices you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



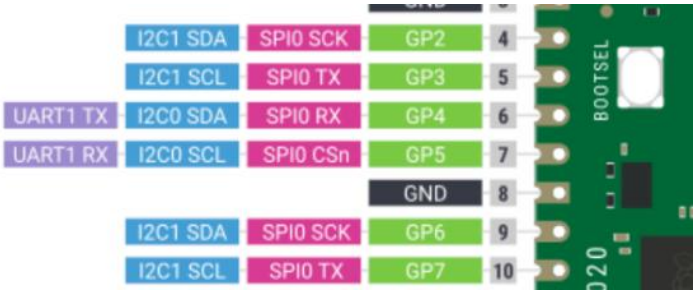
Typical SPI CanSat Uses:
Cameras, Storage cards (e.g. SD cards), GPS modules, WiFi Modems

Exercise 3.1: Connecting to the RFM9x LORA Radio Module using SPI

The RFM9x 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.

1. Connect the power signals on the RFM9x. You will need two cables to connect the VIN and GND pins on the RFM9x 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 RFM9x. 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 RFM9x 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 RFM9x 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 RFM9x, pin GP6 is used in this example.

4. The RFM9x 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 RFM9x, 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 busio libraries to access the SPI interface. We also need the digitalio library for the CS and reset pins and finally we also need the RFM9x radio library provided by Adafruit:

```
import digitalio
import board
import busio
import adafruit_rfm9x
```

3. Now we setup the SPI interface so that we can communicate to the RFM9x. We map the SPI signals to the pin numbers based on how they were connected earlier.

```
spi = busio.SPI(clock=board.GP2, MOSI=board.GP3, MISO=board.GP4)
```

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

```
cs = digitalio.DigitalInOut(board.GP6)
reset = digitalio.DigitalInOut(board.GP7)
```

5. We can now start up the radio. To do this we can call the RFM9x library functions, this will give us an object that we can then use to represent the radio:

```
rfm9x = adafruit_rfm9x.RFM9x(spi, cs, reset, 433.0)
```

   As you can see, the RFM9x() function takes several parameters:
   - The SPI interface to use
   - The CS and reset pins to use
   - Operating frequency: Several different RFM9x are available that operate at different frequencies, 433MHz is recommend to use in the UK as it is part of a free ISM radio frequency allocation. Therefore the operating frequency is set to 433.0MHz

The *rfm9x* object that this function returns is what we shall use for accessing the radio for other parts of the lab.

6. Finally add a message to say that the radio has started up successfully:

```
print("RFM9x radio ready")
```

7. Go back to *code.py* and import the radio module you have just created:

```
import radio
```

8. Run the code and ensure that the *"RFM9x Radio Ready"* message is printed out. If so then your wiring and radio module have been set up successfully, if not check your wiring and the pin assignments in the code.

Exercise 3.3: Radio Data Transmission (CanSat)

1. Now that the radio is set up, we can send a test message. Open *radio.py* and create the following *send()* function that calls the RFM9x library function:

```
def send(message):
    rfm9x.send(message)
```

2. Add the following code within the while loop in *code.py*, just before the *time.sleep()* call:

```
radio.send("Test from CanSat!")
print("Radio message sent")
```

3. Run the code. Check that you still get the *"Radio Ready"* message. If you do not then something has gone wrong in setting up the radio, so first double check your wire connections and then your software from the previous exercise. If all has gone well then the confirmation message should show up in the serial monitor every second to show your message has been sent.

4. Now check that your radio message has either been received by the shared ground station or by one of your neighbours who is running their groundstation code.

5. We shall now extend the transmission message to also contain the temperature and pressure readings. To do this we shall need to include the BMP280 libraries again. Make sure that your BMP280 library is being imported:

```
import bmp280
```

6. Now that the BMP280 is set up we can read the BMP280 sensors in the same fashion as in the previous lab. Write the code required to read the sensors within the `while True:` loop of the last exercise and store them in two variables: `room_temp` and `room_pressure`.

7. We now need to send these values to the radio. Add the following line after the BMP280 readings (still within the *while* loop) to send the sensor readings via string conversions. Replace CANSAT NAME with a name that you can use to identify your CanSat:

```
radio.send("[CANSAT NAME] Temperature: {:.2f} Pressure {:.0f}".format(room_temp, room_pressure))
```

8. Increase the time delay in the loop to 5 seconds to help reduce radio traffic on the shared groundstation.

9. Run the code and check that the received temperature and pressure values are as expected.

Exercise 3.4: Radio Data Receive (Groundstation)

1. As with the transmit, the radio setup of Exercise 4.2 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.
   Open *radio.py* and create the following function:

```
def try_read():
    return rfm9x.receive(timeout=1.0)
```

   This function will check if the RFM9x has received a packet and if it hasn't it will wait for 1 second (set by *timeout*) to see if a packet arrives. It will return the packet data if one has been received, otherwise it returns the value *None*.

2. Whilst in *radio.py* we will add a second function that gets the *"Received Signal Strength Indication (RSSI)"* variable which is measured in decibels. The signal strength is very low and so a typical transmission should have a RSSI of between -40 and -90 dB. Any values lower than this and you should consider upgrading your transmit or receive antenna (N.B. it's much easier to upgrade the groundstation antenna to a YAGI rather than the CanSat's!).

   Add the following function to *radio.py* that fetches the RSSI value from the RFM9x:

```
def rssi():
    return rfm9x.rssi
```

3. The received packet now needs to be read from within *code.py*. First remove any code relating to the bmp280 as the groundstation will not need to use this sensor.

4. Add a variable to count the number of packets received and set it to 0. This needs to be added before the `while True:` loop.

```
packet_count = 0
```

5. Within the loop we can add the code to receive the data. First try to read the data:

```
radio_message = radio.try_read()
```

6. We can then check the result of trying to read the radio and print out the message if a message was received:

```
if radio_message is not None:
    print("Radio RX {:d} {:s}".format(packet_count, str(radio_message, 'ascii')))
    print("RSSI: {:3d}db".format(radio.rssi()))
    packet_count = packet_count + 1
```

   The first *print()* is for the value of the packet counter and the message. The message needs converting into a string using the *str()* function with the ascii parameter.
   The second print prints out the RSSI value.
   The packet counter is then updated.

7. The *sleep()* statement can be removed from the end of the while loop as the *try_read()* function provides the delay.

8. 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.