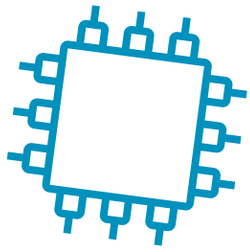
Raspberry Pi Pico W

projects for schools



Arm would like to thank the Raspberry Pi Foundation for their ideas and feedback on this resource. This collaboration is separate to all the great work the Raspberry Pi Foundation is doing with the U.K.'s National Centre for Computing Education, which Arm is also proud to support. To find out more about the NCCE, visit teachcomputing.org

Raspberry Pi Foundation

With updates, 2025

arm school program

Arm Education Media is an imprint of Arm Limited, 110 Fulbourn Road, Cambridge, CBI 9NJ, U.K.

Permissions

Notices

Arm is committed to making the language we use inclusive, meaningful, and respectful. Our goal is to remove and replace non-inclusive language from our vocabulary to reflect our values and represent our global ecosystem.

Arm is working actively with our partners, standards bodies, and the wider ecosystem to adopt a consistent approach to the use of inclusive language and to eradicate and replace offensive terms. We recognize that this will take time. This book may contain references to non-inclusive language; it will be updated with newer terms as those terms are agreed and ratified with the wider community.

Contact us at education@arm.com with questions or comments about this course. You can also report non-inclusive and offensive terminology usage in Arm content at terms@arm.com.

ISBN: 978-1-911531-46-3

For information on all Arm Education Media publications, visit our website at
https://www.arm.com/resources/education/books

To report errors or send feedback, please email edumedia@arm.com

CONTENTS

INTRODUCTION

This book introduces learners to the exciting Raspberry Pi Pico W from Raspberry Pi and explores its use. This cheap yet powerful microcontroller can be programmed in Python. A multitude of devices can be connected to it, including some designed specifically for this microcontroller and some more generic devices. While the book is written around the Pico W, the projects can all be adapted to work with the Pico.

A gradual approach will be taken through this book, taking learners from the basics of installing Python on the Pico W and running their first programs to tackling some challenging projects suitable for older learners who have already developed their programming skills.

A range of additional Computer Science knowledge and skills will be touched upon, including finite state machines, two's complement, and others. Learners will explore the use of a range of Internet services, pushing data to visualization services, pushing data to mobile phones via IFTTT, and analyzing data in Jupyter Notebooks.

Understanding the Raspberry Pi Pico W

Learners often see microcontrollers such as the Pico as delicate, confusing, and scary. While there are elements of delicacy to the Pico, such as the pins being easy to bend, learners should be encouraged not to fear it. It helps to think about the Pico—and any microcontroller—simply as a small computer; all it does is take inputs, calculate the right thing to do, and produce some outputs.

The fact that we use sensors instead of keyboards and mice, and we use LEDs and motors rather than a screen, should not detract from this. Sometimes these inputs will come from external sources such as the HuskyLens, and sometimes the storage will be remote, but learners will be walked through this in accessible steps.

The hardware

Other RP2040 devices are available from a range of manufacturers. This book uses the Raspberry Pi Pico W because it is known, it has wireless networking capabilities, and has been tested with the range of accessories described in the book, including a range of add-ons developed specifically for it by Pimoroni. The Pico W was the 2nd RP2040 board produced by Raspberry Pi foundation. It has since lanched the Pico 2 and the Pico 2 W based on the new 2350 chip. While much of the code in this book will work with the new Pico, at the time of updating this book the Pimoroni libraries for the 2350 were still under development.

GitHub

This book includes some quite large program files. These have all been included within a GitHub repository, or repo, at https://github.com/arm-university/RPi-Pico-projects-for-schools. You can download a zip of the whole repository if you wish, or navigate the individual folders that correspond with the chapters in the book. Some of the files are shortcut links to resources online that may be useful, and these may only work if you download them. While you are welcome to download and use the files as you wish, we always recommend typing in code rather than copying and pasting it, as this creates muscle memory and encourages you to really think about what you are developing.

1. GETTING STARTED

Setting the scene

The Raspberry Pi Pico W is a powerful microcontroller combining a lot of the flexibility and portability of other development boards with a lot of the power of the Raspberry Pi. Through this book you will explore some of this flexibility and power, but before that you need to get to grips with making the Pico do what you want.

To this end, this chapter will introduce you to programming the Pi Pico W in Python both using an interactive shell and then as a complete stand-alone Python program. You will light up both a built-in LED (light-emitting diode) and an external LED and will consider how such a simple technique can be used to control larger amounts of LED lighting. These are the initial concepts involved in smart lighting and further through this chapter you will explore the use of timers for lights. As you work through the book and find out about new techniques of interactivity, you may like to consider how you could build further "smartness" into your lighting creations. Smart lights are not just fun, but encourage the reduction of energy waste and improve sustainability. This links in to Global Goal 11, Moving toward sustainable cities and communities.

Success criteria

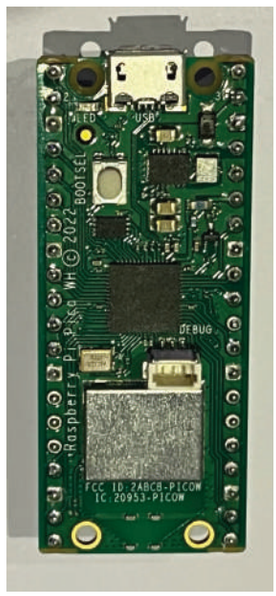
- ☑ Upload the Python environment to the Pico W and test to enable programming.
- ☑ Light the built-in LED from interactive programming mode to demonstrate how outputs can be controlled.
- ☑ Light the built-in LED using a saved program to demonstrate reusability of code.
- ☑ Connect an external LED and control it using a push button, turning it off on a timer.
- ☑ Connect an external LED via a transistor to an external power supply and build a small-scale, low-power, energy-efficient lighting circuit.

1

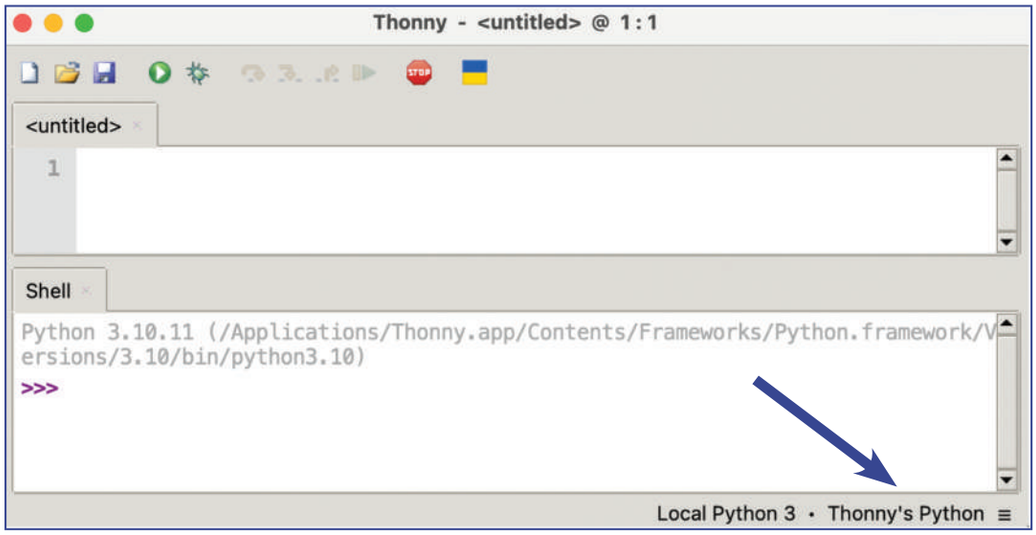
One of the main differences between a microcontroller and a computer is you can program a computer by typing a program into it through an *integrated development environment (IDE)*—a tool that combines all your development tools under a common interface—and running it directly, but you can't type a program directly into a microcontroller. Instead, we need to use some sort of intermediary—in this case we will use our computer and an IDE, Thonny, to program the Pico with a Python interpreter and then the IDE to interact with the Pico from the computer.

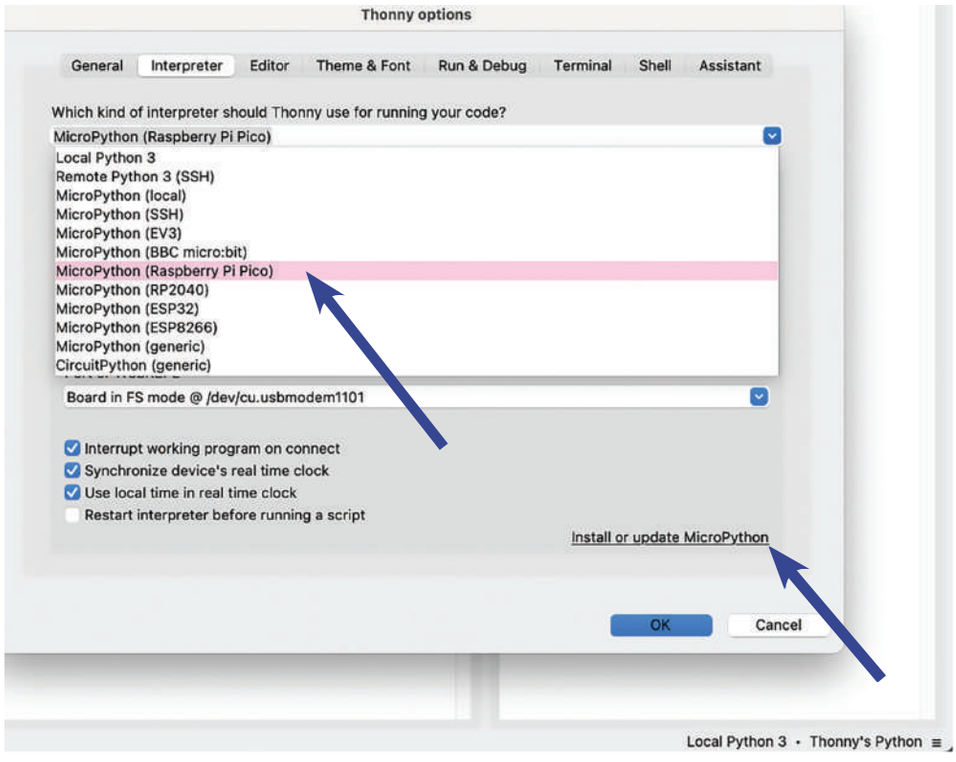
Thonny can be downloaded from https://thonny.org—install the latest version available. Thonny will work on Windows, Mac, Linux, or even on a standard Raspberry Pi.

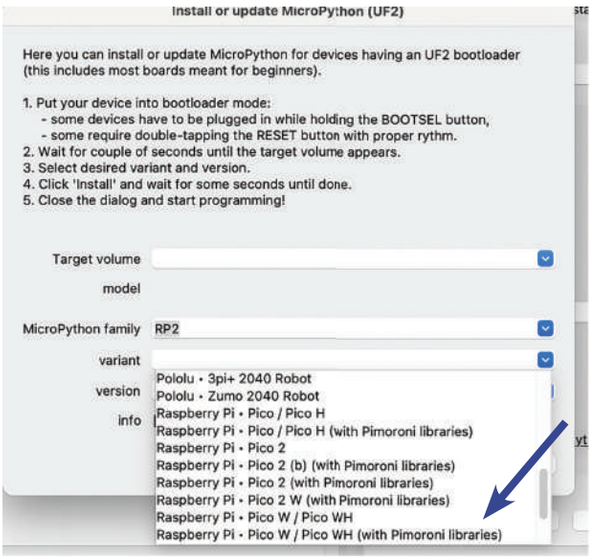
After installing Thonny and running it, you need to connect your Pico in mass storage mode (that is, it can be accessed like a USB storage device or memory card). To do this, hold the white BOOTSEL button and plug the Pico into your computer using a USB cable.



When in BOOTSEL mode, you can click on the Python version at the bottom of the Thonny window, choose MicroPython, then click “Configure interpreter”, and follow the instructions to install the MicroPython firmware onto the Pico. Alternatively, if Thonny is not giving you the option, you can download the firmware from micropython.org and drop it onto the Pi drive in Windows Explorer or Mac Finder. When completed, this will put the Python interpreter itself onto the Pico W. If you unplug the USB from your Pico W and then replug it in (not in BOOTSEL mode), you can choose MicroPython in Thonny.







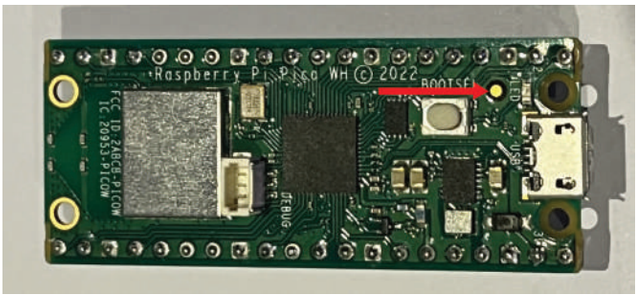
PRO *TIP*

If you are not using the Pico W, you will need to select the relevant device at the "install or update MicroPython (UF2)" stage shown here.

2

When you first started using Python, you may have interacted with it on a command line or by typing directly into a window such as the IDLE (Integrated Development and Learning Environment) Shell. The Shell section of Thonny (the lower panel) is running in interactive mode. If you type these lines into the Shell one at a time you will be able to light and turn off the built-in LED.

```
>>> import machine
>>>
>>> led = machine.Pin("LED", machine.Pin.OUT)
>>> led.value(1)
>>> led.value(0)
>>>
```

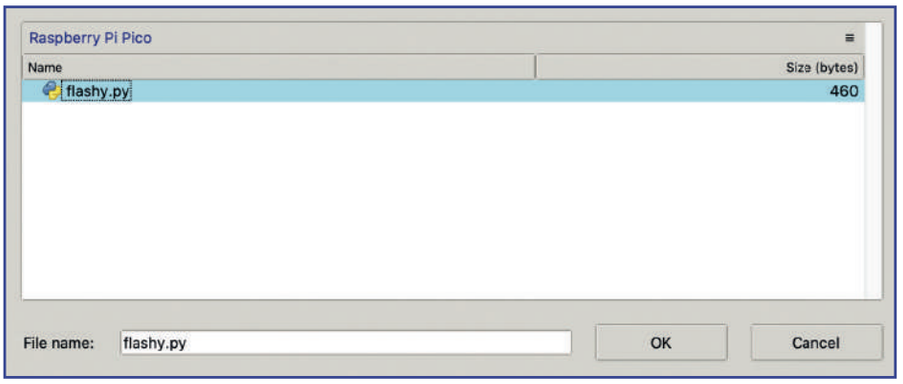


PRO *TIP*

If things get a bit confusing in interactive mode, there is a red STOP button at the top of Thonny which will reset the interpreter.

3

No doubt when you program "normal" Python on your computer, you have moved past using the Shell and now save your files as .py files. This aids reusability, and we can do the same thing in MicroPython with Thonny. Instead of typing in the bottom (Shell) section, type your code in the top (code) section of Thonny and press save when ready.



Because you are **not** in BOOTSEL mode, you are now able to save directly to the Pico. When you are given the option, choose to save in the Pico rather than your local computer and press the green **Run** arrow button.

Type in or load the **flashy** program and run it. The built-in LED should flash on and off at 2-second intervals. Let's explore the code a little:

```
1  from machine import Pin
2  from utime import sleep
3
4  # Constants
5  ON = 1
6  OFF = 0
7  SLEEPTIME = 2  # Number of seconds delay
8
9  # set up the LED PIN
10 led = Pin("LED", Pin.OUT)  # Set up onboard LED pin
11
12
13 while True:
14     led.value(ON)
15     sleep(SLEEPTIME)
16     led.value(OFF)
17     sleep(SLEEPTIME)
18
```

PRO *TIP*

Don't worry if you forget to use utime instead of time—MicroPython will use the right one anyway.

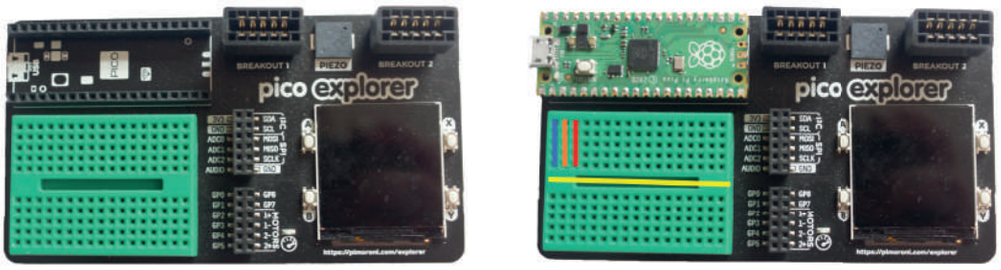
- The **machine** import gives Python access to the Pico. **utime** works very similarly to the **time** module in desktop Python, and in this case we will use it to "pause" execution.
- We set constants for **ON** and **OFF** to aid readability of the code.
- We also set a constant for the delay we want between flashes—in this case 2 seconds.
- We create a **Pin** object called **led**. We give it the ID "LED" and define as an **OUT** pin (for output).
- We then enter a forever loop (`while True`) and turn the **led** value to **ON**, sleep for 2 seconds, **OFF**, sleep for 2 seconds, then repeat.

PRO *TIP*

Constants are named areas of memory that don't change during the program runtime. Using constants is good practice in all programs as they reduce the chance of mistakes happening when values need changing, but in microcontrollers they are often used to reduce the amount of memory used as well.

4

The little LED built into the Pico isn't going to do much good as a bedside light, never mind any sort of home smart lighting. We need to be able to power a much brighter LED. External LEDs require building a circuit, and the Pico Explorer board gives us a handy breadboard to be able to build circuits without breaking out the soldering iron. Carefully plug the Pico into the Pico Explorer board, making sure not to bend the pins.



The columns of holes (e.g. the blue, orange, and red lines) are joined together underneath with a conductive strip; this allows us to connect the legs of components by just pushing them into connected holes. The columns either side of the middle "split" (in yellow) are not connected so we can actually connect quite a few components!

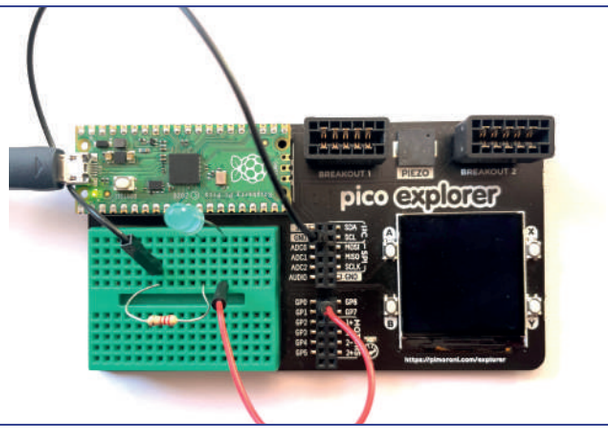
The Pico outputs at 3.3 V. However, an LED may struggle with such a voltage and may need a small resistor to reduce the voltage. (If you think about the electricity as water flowing down a pipe, *voltage* is like the pressure of the water. If something gets in the way, such as a radiator, the water pressure will reduce. Similarly, if something interrupts the flow of the electricity, in this case a resistor, then the voltage will be reduced.) Omitting the resistor can at best reduce the lifespan of the LED, and at worst can actually cause a fire. By using the datasheet of the LEDs you buy, you can calculate the ideal resistor for your LED at 3.3 V. You may not have these details, so for this project we will use a 220 Ω resistor—this is more than sufficient. Power will flow from the GPIO pin (use pin 0, labelled as "GP0" on the Pico Explorer board) into the LED anode (the long leg), out of the LED cathode (the short leg), through the resistor and back to ground. Electricity flows from positive (provided by the GPIO pin in this case) to ground (often shortened to GND). Every circuit needs a return path to ground.

PRO *TIP*

LEDs are directional or polarized. If your LED doesn't light up, try turning it around! If the legs are cut or bent you can identify the cathode as there should be a flat point on the LED base.

PRO *TIP*

You may also use a online calculator such as Resistor Calculator https://ohmslawcalculator.com/led-resistor-calculator.



5

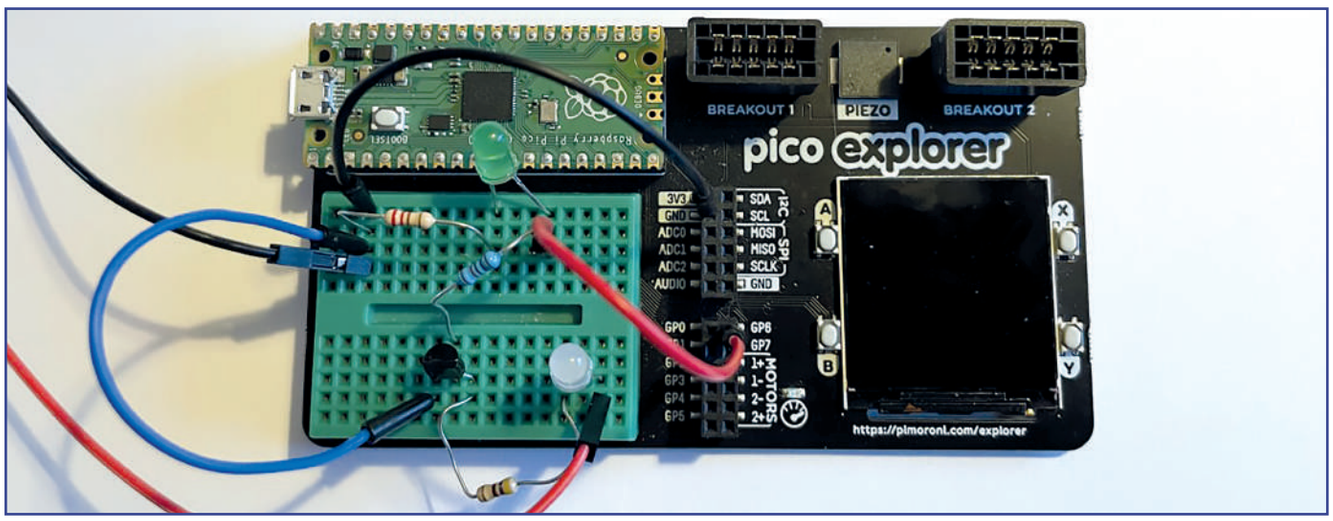
Our smart lighting still has some problems, however; the Pico is run from USB, which gives it a maximum of 3.3 V and (more importantly) 500 mA (half an amp). Components run at certain voltages, so the Pico pushes 3.3 V (and we reduce it with resistors if needed), but the current (amps) is the payload. A component such as an LED will draw as much current as it needs to operate, and if it can't get enough, then it won't operate. The Pico already uses a chunk of the available current and the Pico Explorer board will use a chunk more—meaning that we are not going to have a tremendous amount of current left to run our LEDs.

We can use a transistor—one of the most important inventions of the 20th century—to allow us to connect up our LEDs to an external power supply. In this project we are using an NPN transistor, coded PN2222A. A transistor joins an external power supply running through its collector pin, via anything it needs to power, and back to ground via its emitter pin. Normally this circuit is disconnected, but if it receives a small current on its base pin it will make the connection and allow the (larger) power to flow. This means we can power an external LED from an external power source when the Pico tells us to, but without using lots of the current available to the Pico from the USB connection.

PRO *TIP*

There are a variety of transistors available. The PN2222A transistor is popular. This is known as an *NPN transistor*. The circuit here assumes you have the flat face of the transistor pointing toward you (down on the image). If you have a different transistor model the pins may be in a different order so always check the datasheet!

In this case we will use a 9 V battery for the external power source. Connect the red wire (positive) from the 9 V battery clip to the anode (long) leg of another external LED, and the cathode (short) leg of the LED via a resistor to the collector pin of the transistor. The resistor value should really be calculated, but you can search for "normal" ranges on the Internet. An ultrabright blue LED will need about 290 Ω, so we used 470 Ω to be safe on our normal green LED. If you don't have a high enough value, you can connect multiple lower values (such as 2 × 200 Ω resistors) together end to end.



The emitter pin is connected to the same ground "column" as the ground for your original LED (the blue wire in the image), as is the black (negative) wire from your 9 V battery clip. This is the externally powered circuit connected, but to turn it on we need to send a current from the Pico to the base pin. We can do this by linking a resistor between the positive side of the original external LED (which in itself is connected to GP0) and the base pin. It doesn't take much voltage to turn on the transistor and enable the circuit, so a 10 kΩ resistor is fine.

PRO *TIP*

The PN2222A transistor will allow you to transfer a maximum of 1 A—this will run a decent number of LEDs but it will need to be cooled using a heatsink to run at that level for any time. If you need to run more power-hungry devices, such as more LEDs, motors, fans, and the like, it would be worth exploring more capable (and larger/more expensive) transistors, or even some more capable devices called MOSFETs which often allow higher currents.

6

A flashing LED is of course pretty useless as an opportunity for us to replace or augment our home lighting with something more efficient. The Pico Explorer board has some handy buttons we can use to start a timer, but unfortunately this needs a bit more setting up. If you did not install the Pimoroni version of MicroPython shown earlier in this chapter, you will need to do this to use the Pico Explorer board. If you visit the Pimoroni site you will be able to download a custom MicroPython firmware for the Pico Explorer. You will need to reboot your Pico in BOOTSEL mode again, and drop the UF2 file onto it to install this new firmware.

```
import picoexplorer as explorer
from machine import Pin
from utime import sleep

# Button and LED Setup
button_a = Pin(12, Pin.IN, Pin.PULL_UP)

led = Pin(0, Pin.OUT)  # Set up external LED pin

# Constants
ON = 1
OFF = 0
TIME_FOR_LIGHT = 20  # Number of seconds the LED will stay on

countdown = 0  # Countdown timer variable

while True:
    # Check if button A is pressed
    if button_a.value() == 0:  # Button is pressed (PULL_UP logic)
        countdown = TIME_FOR_LIGHT  # Start countdown
        led.value(ON)  # Turn LED on

    # Countdown logic
    if countdown > 0:
        countdown -= 1
        sleep(1)  # Wait for 1 second
    else:
        led.value(OFF)  # Turn LED off
        sleep(0.1)  # Short delay for loop stability

```

This code will set up a timer for the LED to be on when the button is pressed. Let's explore the code:

- The **picoexplorer** module is imported, as is just the **sleep** function of **utime**.
- A constant is set up to set how long the LED stays on for and a countdown is initialized to 0.

- In the forever (`while True`) loop the button state is checked. If it is pressed, the LED is lit and the countdown is started, then a 1-second sleep is triggered.
- Every loop around the countdown is checked. If it is still above 0, then it is decremented and a 1-second sleep is triggered. Otherwise, the LED is turned off and a one tenth of a second sleep is triggered.
- It is essential that there is some sleep time between the times you check the button. Otherwise, as the Pico is far faster than your finger, it may be detected multiple times.

Testing

It is easy to miscount times when you are using a countdown timer. Use a stopwatch to check the LED stays on for the 20 seconds you requested.

Stretch tasks

- This project so far would be a great "timed" light for your bedroom and will use a lot less electricity than using your main light bulb. It can be quite disturbing, however, when a light suddenly turns off. Adjust the program so that the LED flashes shortly before turning off so as to enable you to press the button again and reset the timer.
- If you use this as a nightlight while you fall asleep, it is not necessary to have bright light on all the way through. Connect three LEDs using transistors to an external power source. Have each LED turn off at different points in the countdown so that the total amount of light gets dimmer over time.
- Using a 9 V battery allows us to run far more LEDs than just using the Pico, but the number is still limited. Explore how many LEDs you need to form a suitable nightlight in your bedroom, and investigate how long a 9 V battery will last running this number of LEDs. Compare this to the amount of power a 9 V AC/DC adaptor will use and draw conclusions as to which might be the better choice environmentally.

Final thoughts

LEDs are far more efficient than traditional light bulbs because they use a lot less power to produce a similar amount of light. The light output of a bulb is measured in Lumens, with a normal old-fashioned 60 W incandescent bulb giving out approximately 840 Lumens. Watts is a measurement of how much energy is being used—1 W is 1 Joule per second—a 60 W bulb uses ½ amp of current at 120 V (U.S. system) or about ¼ amp at 230 V (British system). For comparison, an equivalent LED bulb might give out approximately 800 Lumens, almost the same, but uses only 13 W—less than a quarter of the power. Not only is the lower power production good for your bank balance and the environment, but there is also far less wastage, with a traditional bulb lasting on average 0.9 years and the equivalent LED bulb lasting nearly 23 years! For us to reach Global Goal 11, we need to use less energy, and what we do use needs to last longer than it does now.

2. DISPLAYING ENVIRONMENT DATA

Setting the scene

Our local environment can have a huge impact on our general well-being. Road traffic pollution is linked to illnesses and deaths all around the world. The average global temperature has been slowly rising for decades. The amount of CO_2 in a room has a negative effect upon our productivity, as well as being a sign that there are too many people in the room with not enough circulation, something that became all the more apparent in the COVID-19 pandemic circa 2020. It is clear that urgent action is needed to combat climate change and meet Global Goal 13, Climate action, and to develop that we need to expand our knowledge, something that will be explored in this chapter.

The Raspberry Pi Pico W, along with the Pico Explorer, has a fantastic ecosystem of breakouts which allow us to plug in and connect to a range of inputs and outputs, including many types of sensors. In this chapter, you will be using the BME680 breakout to monitor the environment around you, and using the display screen on the Pico Explorer to display the data.

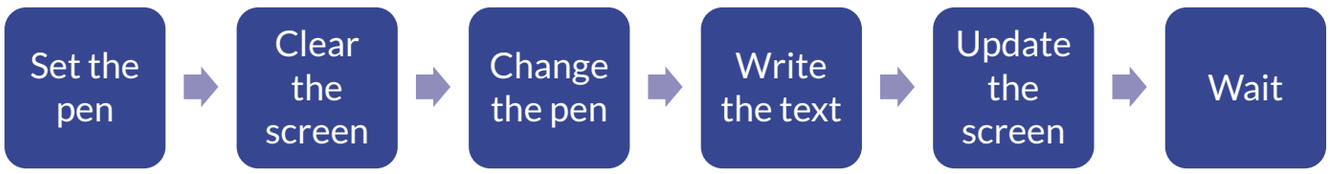
Success criteria

- ☑ Use the OLED (organic light-emitting diode) display on the Pico Explorer to display test data.
- ☑ Connect the BME680 environment monitor and display data on the command line.
- ☑ Display the environment data on the OLED.
- ☑ Design an experiment to test memory retention depending on air quality.

1

The Pico Explorer has a 240 × 240 pixel screen which is tremendously useful for displaying information. If you explore the examples that come with the board, you will see it is quite vibrant and reactive. You will find a link to these examples in the GitHub repository.

For this project we will just be displaying text. The process is as follows:



The `display.create_pen()` function allows us to create pens with predefined colors and uses combinations of red, green, and blue (in that order), which can range from 0 (none of that color) to 255 (as much of that color as possible). For example, `BLUE = display.create_pen(0, 0, 255)` creates a new pen called **BLUE**, which is bright blue. This is used both for the writing text and for clearing the screen. Hence `display.set_pen(BLUE)` followed by `display.clear()` will appear to clear the screen and set a bright blue background. You can look up common colors by searching the web for "RGB Decimal Chooser".

The **text** function is used for writing on the screen. It takes five parameters, the text that will be written (a string), the *x*- (horizontal) and *y*- (vertical) position to start writing, and the maximum width of the text (at which point it will wrap). Hence, `display.text (quote, 20, 20, 200, scale = 2)` will print the text stored in **quote** 20 pixels in from the top left corner and, as the screen is 240 pixels wide, will wrap at an even margin. The scale = 2 parameter scales up the font size. The example code below will write some text on a blue background in white text. Note that *x*- and *y*-coordinates work in the same way as mathematical coordinates, although computer screens start at the top left corner.

PRO *TIP*

You can create a loop to try varying colors and *x*- and *y*-coordinates and figure out what looks right for your purpose.

```
import time  # To enable delays
from picographics import PicoGraphics, DISPLAY_PICO_EXPLORER, PEN_RGB332

# Initialize display
display = PicoGraphics(display=DISPLAY_PICO_EXPLORER, pen_type = PEN_RGB332)
WIDTH, HEIGHT = display.get_bounds()  # Get screen dimensions
BLUE = display.create_pen(0, 0, 255)
WHITE = display.create_pen(255, 255, 255)

# Define the text to display
quote = (
    "1925 IBM Manual: All parts should go together without forcing. "
    "You must remember that the parts you are reassembling were disassembled by you. "
    "Therefore, if you can't get them together again, there must be a reason."
)

while True:
    # Set pen to blue and clear the screen
    display.set_pen(BLUE) # Blue
    display.clear()  # Clear the screen with the current pen color

    # Set pen to white and display the text
    display.set_pen(WHITE)  # White
    display.text(quote, 20, 20, 200, scale=2)  # Display text at (20, 20) with scale 2
    # Update the screen to apply changes
    display.update()
    # Short delay to control loop speed
    time.sleep(0.1)  # 100ms delay

```

2

The BME680 breakout can be plugged straight into BREAKOUT 1 on the Explorer base. The Pimoroni firmware comes with all of the drivers needed so we can read it over the I2C bus quite simply. The *I2C* or inter-integrated circuit bus allows multiple devices to connect via the same bus without using up too many pins on the microcontroller. Many sensors (including the BME680) support it and the Explorer base connects this up to the breakouts for us.

Once it has been configured, the BME can be read to identify temperature (in Celsius), pressure (in Pascals), humidity (as a percentage) and air quality (measured as a resistance in Ohms). The code below will perform this configuration and then output these readings once a second. You may find it odd that air quality is measured in resistance. This is due to gases in the air reacting to a sensor surface and changing its resistance, which is what we can measure in Ohms.

Let's explore the code:

```
import time
from breakout_bme68x import BreakoutBME68X
from pimoroni_i2c import PimoroniI2C
from pimoroni import PICO_EXPLORER_I2C_PINS
from picographics import PicoGraphics, DISPLAY_PICO_EXPLORER

# set up the hardware
display = PicoGraphics(display=DISPLAY_PICO_EXPLORER)
i2c = PimoroniI2C(20,21)
bme = BreakoutBME68X(i2c, address=0x76)

# set up pen
WHITE = display.create_pen(255, 255, 255)
RED = display.create_pen(255, 0, 0)


# Main loop to read sensor data
while True:
    # Read sensor values as tuple
    temperature, pressure, humidity, gas_resistance, *_ = bme.read()

    # Print each reading
    print("Temperature (°C):", temperature)
    print("Pressure (Pa):", pressure)
    print("Humidity (%):", humidity)
    print("gas_resistance (ohms): ",gas_resistance)

    # Wait for 1 second before the next reading
    time.sleep(1.0)

```

- The imports include time (for the delay between readings), the BME68x breakout (there are other chips in the family with similar names), the Pimoroni I2C interface, and picographics imports **PicoGraphics** and `DISPLAY_PICO_EXPLORER`.
- The pins used by the Explorer to connect to I2C are defined (SDA and SCL, 20 and 21 respectively). If you turn your Explorer base upside down you can see these identified.
- The I2C and the BME are created with the necessary parameters.
- Two pen colors are set up.
- A continual loop is created, and looped once per second. Within this, the BME680 is read (which produces a tuple) and the readings that are wanted are stored in variables. Storing a value to *_ means we will not look at that value—the BME provides some things we don't need.
- The readings are output to screen.

3

To make the sensor more able to execute independently, we shall combine the last two programs to output the BME data onto the screen of the Explorer:

```
import time
from breakout_bme68x import BreakoutBME68x
from pimoroni_i2c import PimoroniI2C
from pimoroni import PICO_EXPLORER_I2C_PINS
from picographics import PicoGraphics, DISPLAY_PICO_EXPLORER

# set up the hardware
display = PicoGraphics(display=DISPLAY_PICO_EXPLORER)
i2c = PimoroniI2C(20,21)
bme = BreakoutBME68x(i2c, address=0x76)
# create pens
BLUE = display.create_pen(0, 0, 255)
WHITE = display.create_pen(255, 255, 255)


while True:
    temperature, pressure, humidity, gas_resistance,-,-,- = bme.read()# Read sensor values as tuple
    # Set pen to blue and clear the screen
    display.set_pen(BLUE) # Blue
    display.clear()  # Clear the screen with the current pen color

    # Set pen to white and display the text
    display.set_pen(WHITE)  # White
    print("Temperature (°C):", temperature)
    print("Pressure (Pa):", pressure)
    print("Humidity (%):", humidity)
    print("gas_resistance (ohms): ",gas_resistance)

    display.update() #Update the screen to apply changes
    time.sleep(0.1)  # 100ms delay

```

Explore the sensor and make notes of what is “good” and what is “bad” in your environment. We found clear air to be >60,000 while <45,000 was found by breathing heavily on the sensor.

PRO *TIP*

If you are struggling to see the display while manipulating the sensor, you might like to re-enable the `print()` commands to output to your computer screen.

4

Visual displays are good for accurate detail, but it is unlikely that you will be looking at one regularly during the day. The Pico Explorer has a buzzer attached to it. If you put a jumper wire between a GPIO pin (such as GP0) and the “audio” pin and add buzzer to the Pimoroni imports, you can use the `BUZZER.set_tone()` function to make the buzzer sound. The code below has a **warning** function which sounds when the gas quality gets below a certain level.

```
import time
from breakout_bme68x import BreakoutBME68X
from pimoroni_i2c import PimoroniI2C
from pimoroni import PICO_EXPLORER_I2C_PINS, Buzzer
from picographics import PicoGraphics, DISPLAY_PICO_EXPLORER

display = PicoGraphics(display=DISPLAY_PICO_EXPLORER)# Set up the hardware
i2c = PimoroniI2C(20,21)
bme = BreakoutBME68X(i2c, address=0x76)

BLUE = display.create_pen(0, 0, 255)# set up pens
WHITE = display.create_pen(255, 255, 255)

BUZZER = Buzzer(0)  # Create a buzzer on pin 0
GASTRIGGER = 45000  # Threshold for gas resistance warning

def warning(): # Warning function
    """Produce a warning tone using the buzzer."""
    for _ in range(10):  # 10 beeps
        BUZZER.set_tone(600)  # 600 Hz tone
        time.sleep(0.1)
        BUZZER.set_tone(-1)  # No sound
        time.sleep(0.1)
while True: # Main loop
    temperature, pressure, humidity, gas_resistance, _, _, _ = bme.read()# Read sensor values as tuple
    display.set_pen(BLUE)
    display.clear()
    display.set_pen(WHITE)
    print("Temperature (°C):", temperature)
    print("Pressure (Pa):", pressure)
    print("Humidity (%):", humidity)
    print("gas_resistance (ohms): ",gas_resistance)

    if gas_resistance < GASTRIGGER:    # Check gas resistance threshold
        warning()
    time.sleep(1.0)  # Sleep for 1 second
```

PRO *TIP*

Pimoroni's GitHub repository has an example called `noise.py`, which will show you how to play a variety of sounds—and in fact a whole song—on the Pimoroni buzzer.

5

Air quality is known to affect productivity. Extend your program to sound a (different) buzzer every 30 minutes (1200 seconds). At this point, complete a memory exercise such as a pairs game at https://www.helpfulgames.com/subjects/brain-training/memory.html and record how well you do and the air quality at the time. Do you notice any differences over time in terms of how you complete the game and what the air quality reading is?

Testing

It is likely that your readings will be fairly stable. There is nearly always a little variation due to interference and environmental factors; watch your sensor and identify a range of values which you could define as "stable". You can alter the temperature by holding the sensor in your hands, and the air quality by breathing on the sensor. A piece of orange peel (or similar) over the sensor will show a change of humidity, and waving the sensor in the air will vary the pressure a little.

Stretch tasks

- Alter the code so that the screen turns blue at a given cold temperature and red at a hot temperature. What you choose as hot and cold will of course vary depending on what you are used to!
- Capture a press of **Button A** and start keeping an average gas quality reading from that point until you press **Button B**. Record this along with your memory test results.
- When the warning buzzer is sounding, allow **Button X** to silence it. Ensure it will not sound for the next three minutes, but will then start sounding again if the air quality is still low.
- Explore other ways you could adjust air quality—where could you put your sensor that is likely to have better or lower air quality than where you are working?

Final thoughts

Being aware of your own surroundings and the environment you work in can have a huge impact. The Smart Citizen project (https://smartcitizen.me/) is tackling Global Goal 13, and has thousands of sensors around the world monitoring the environment using IoT devices not dissimilar to the one you have just built. Crowdsourcing air quality measurements, for example, can put pressure on governments to work to develop lower pollution levels. What other groups, local, national, and international, do you think might be interested in using the kind of readings you are collecting to put pressure on those in charge?

3. ANALYZING ENVIRONMENT DATA

Setting the scene

In the previous chapter you were asked to do some rudimentary analysis of data; in this chapter you will be learning some more formal techniques. You will log data from the environment sensor to a Comma Separated Value (CSV) file, copy that file to your computer, and use the de-facto industry-standard Jupyter Notebook to look at the data and explore it in more detail.

There is a detailed guide to using Jupyter Notebooks here: https://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/

This exploration of environment data in large data sets allows data scientists to build powerful models of climate changes over time and, ultimately, allows a detailed picture of environmental changes. These models are then used to raise awareness of climate change and, hopefully, guide future policy to help us meet Global Goal 13, Climate action.

Success criteria

- ☑ Format data into a structure and log it to a file.
- ☑ Copy the data from the Pico to your computer.
- ☑ Install Jupyter Notebook.
- ☑ Open the data file in Jupyter Notebook and explore it.

1

The Pico has a small amount of on-board memory (2 MB or 4MB on the 2nd generation boards). If you want to log large amounts of data you might like to explore an external SD card module (although this may make attaching the sensor a bit harder at this point)—this will be described in future modules.

However, as we are storing relatively small amounts of data, we will restrict ourselves to the on-board flash memory. The code below concatenates the sensor data into a single string, adds a new line, and writes it to a text file called **environment.csv**.

```python
import time
from breakout_bme280 import BreakoutBME280
from pimoroni_i2c import PimoroniI2C
from pimoroni import PICO_EXPLORER_I2C_PINS, Buzzer
from picographics import PicoGraphics, DISPLAY_PICO_EXPLORER

display = PicoGraphics(display=DISPLAY_PICO_EXPLORER)# Set up the hardware
i2c = PimoroniI2C(20,21)
bme = BreakoutBME280(i2c, address=0x76)

BLUE = display.create_pen(0, 0, 255)# set up pens
WHITE = display.create_pen(255, 255, 25)

BUZZER = Buzzer(0)
# Function to produce a warning tone
def warning():
    for _ in range(10):  # 10 beeps
        BUZZER.set_tone(600)  # 600 Hz
        time.sleep(0.1)  # 1/10 second
        BUZZER.set_tone(-1)  # No sound
        time.sleep(0.1)  # 1/10 second

# Constants
XPOS = 20  # X position for text
WIDTH = 200  # Width for text wrapping
GASTRIGGER = 35  # Threshold for air quality warning
FILENAME = "/environment.csv"  # File to store data

# Prepare file
with open(FILENAME, "w") as f:
    pass  # Create or clear the file

# Main loop
while True:
    # Open file for appending
    with open(FILENAME, "a") as f:
        # Clear screen
        display.set_pen(BLUE)  # White pen
        display.clear()

        # Read data from the sensor
        display.set_pen(WHITE)  # Black pen
        temperature, pressure, humidity, = bme.read()

        # Display data on screen
        display.text(f"Temperature (°C): {temperature:.2f}", 5, 20, scale=2)
        display.text(f"Pressure (hPa): {pressure:.2f}", 5, 40, scale=2)
        display.text(f"Humidity (%): {humidity:.2f}", 5, 60, scale=2)
        #display.text(f"Gas (Ohms): {gas_resistance:.2f}", 5, 80, scale=2)
        display.update()

        # Write data to the file
        file_string = f"{temperature},{pressure},{humidity}\n"
        f.write(file_string)


        # Update the screen
        display.update()

        # Trigger warning if air quality is poor
        if temperature > GASTRIGGER:
            warning()

    # Delay before the next loop iteration
    time.sleep(1.0)
```

A CSV file allows us to organize data with one “set” of data per row, with elements being comma separated. This is a standard file format that will display nicely in most spreadsheet programs. The images show a CSV file as it would appear in a spreadsheet package, and how the raw file looks in a text editor. Note that this is merely an example and not data created by the program above.

	A	B	C	D
1	timestamp	x	y	z
2	0001	8	8	10
3	0002	8	1	1
4	0003	1	5	4
5				

```
timestamp,x,y,z
0001,8,8,10
0002,8,1,1
0003,1,5,4
```

You may notice the program opens the file in “w” mode (write mode) and then closes it again, and then once in the loop it opens it in “a” (append) mode. The

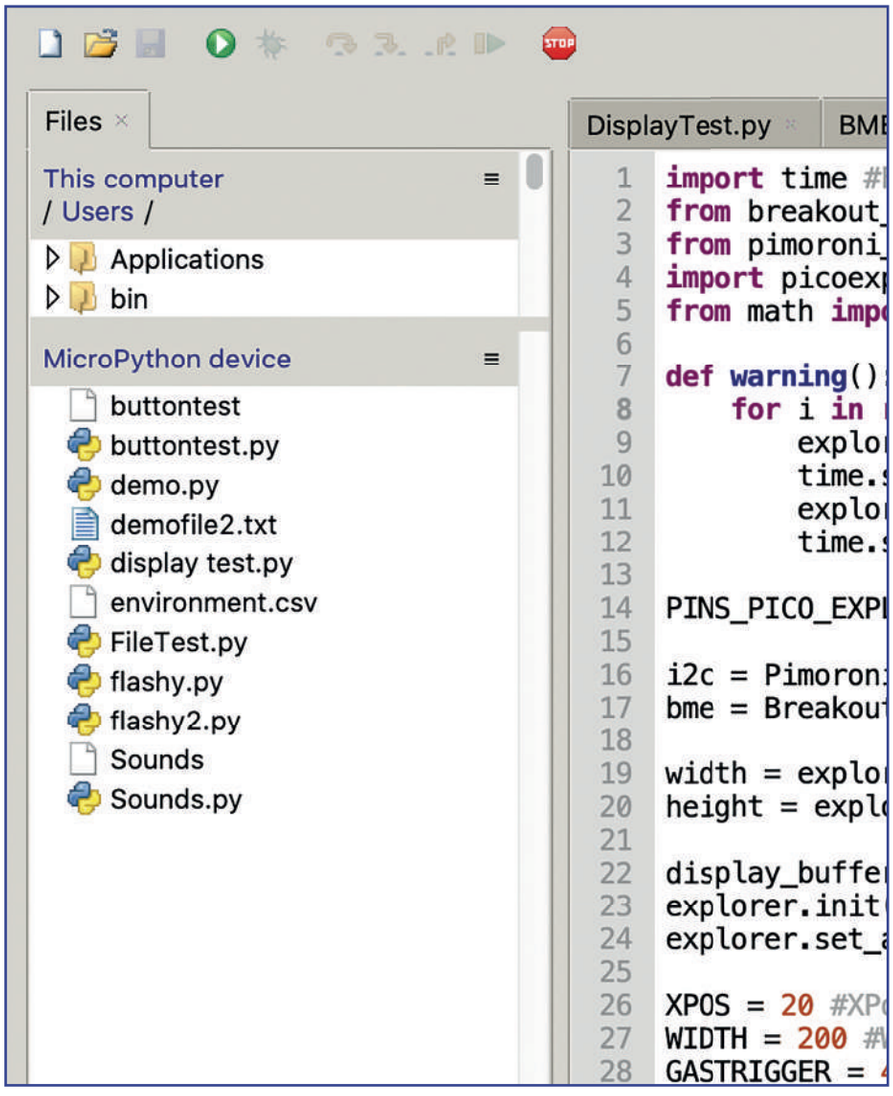
initial open will ensure the file is blank (getting rid of old data). Appending then allows new data to be added to the end of the existing data within this execution of the loop. The file is closed and opened again each time around to ensure that the data is fully written. Possibly excessive in this case, but good practice in general. You may be aware of other file modes in Python, such as "r" for read, although we won't be using them in this project.

PRO *TIP*

Circuit Python tends to get a bit upset if you concatenate (join together) too many things at once. If this happens, just break it down to multiple steps as we have done. You may need to unplug and replug your Pico to get everything working again.

2

The file you create is stored on the internal flash memory, which isn't immediately available to your computer (even if you boot up in BOOTSEL mode). Luckily, Thonny comes with a handy tool to be able to access the file. If you select the **View** menu there is a **Files** option. When you select this you will see a files list come up on the left-hand side of your screen. The image shows this in Thonny. You can right-click the **environment.csv** file and **Download to...** your home directory (and then copy it wherever you need it).

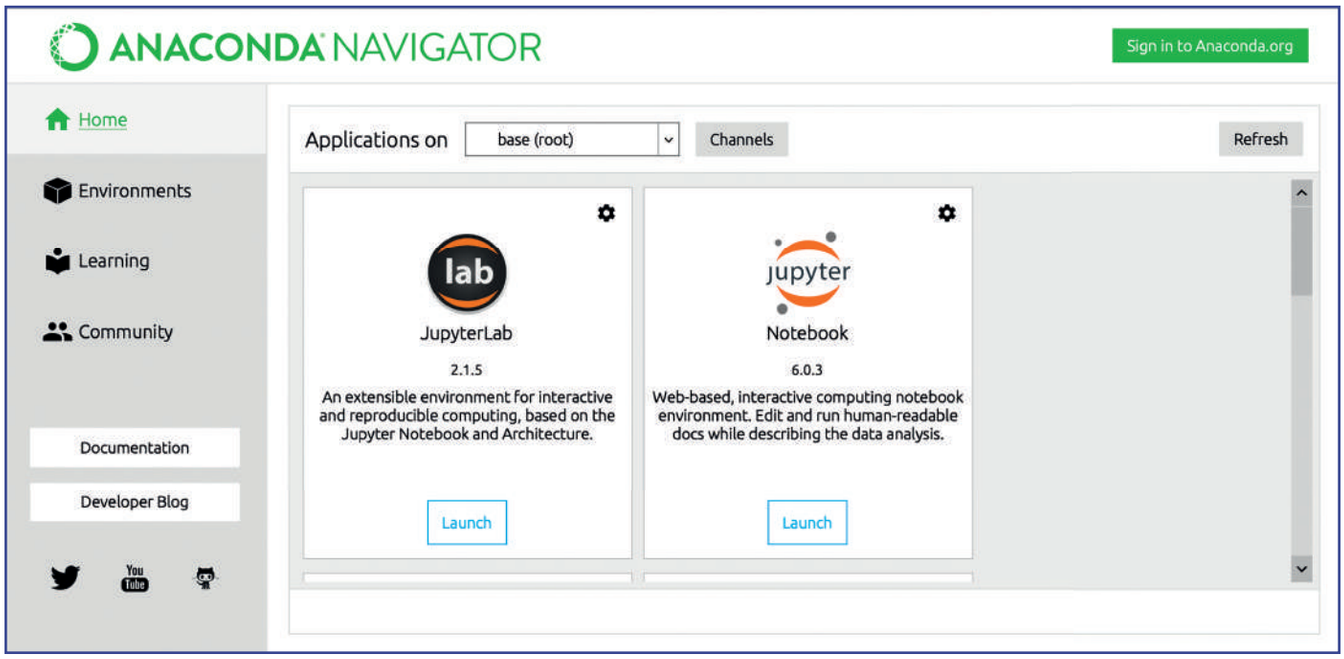


PRO *TIP*

If the copy process doesn't seem to be working, try using the yzuv button on Thonny first to free up the Pico.

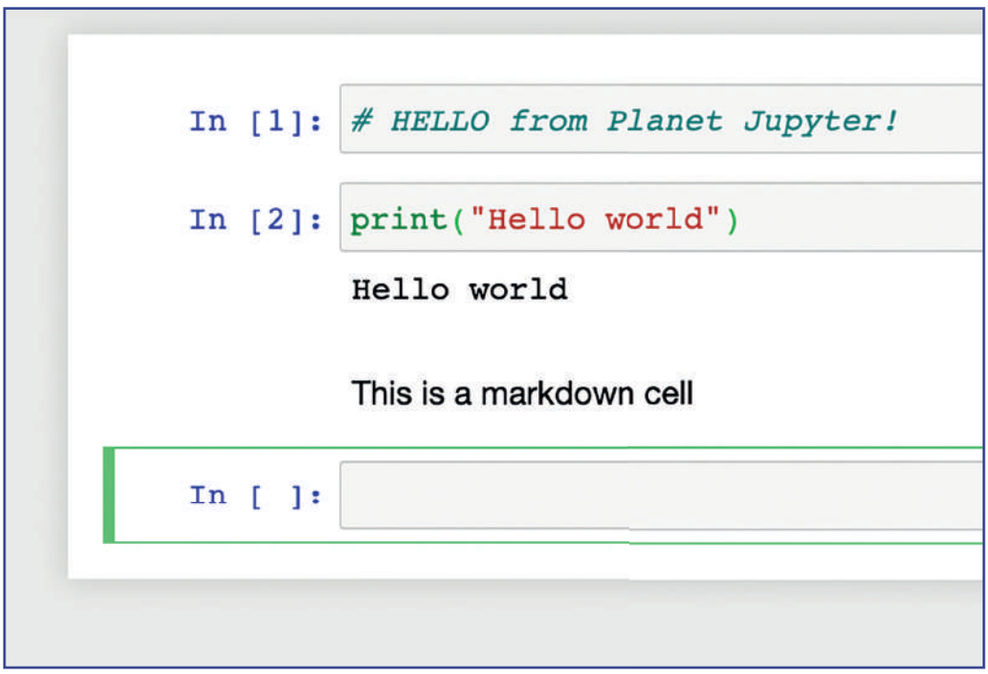
3

Notebooks in Jupyter are an excellent way of combining explanatory text (in Markdown format) with snippets of code (often in Python, although many other languages are supported). If you have Python (and, more importantly, Pip) installed, then the simplest way to install Jupyter Notebook is to open a command prompt and type `!pip install jupyter` and start it by typing `jupyter notebook` from the command line. If this doesn't work (often the case on Windows computers), you may like to download and install Anaconda and launch from Anaconda Navigator.



When Jupyter Notebook loads up, you will be given a list of files. You may want to create a folder to work in. Create a new notebook and explore. Cells can be either code (Python) or Markdown (text) and can be **Run** to run the code (or render the Markdown).

Make sure to copy the CSV file into the same folder location as your notebook for the next step.



4

We can explore the CSV file data using the excellent Pandas library. Pandas is an industry-standard wrapper for MatPlotLib (used for visualization of data) and NumPy (used for mathematical operations) and is commonly used to simplify access to these powerful libraries.

```
In [3]: import pandas as pd
```

Pandas can then read the CSV into its own internal data structure, although as you can see, our lack of headings causes a problem. With a notebook in Jupyter we can edit the code cell and re-run it to alter the output. We can use the Markdown cells to keep some handy notes as we go along.

```
In [1]: import pandas as pd

In [2]: data= pd.read_csv("environment.csv")
        data.head()
```

Out[2]:

	19.51074	100787.1	61.80923	410444.6
0	19.53587	100785.9	61.80641	10254.170
1	19.53084	100787.7	61.74796	9232.594
2	19.53838	100786.3	61.70413	10364.830
3	19.55095	100793.8	61.66745	11742.580
4	19.55346	100786.1	61.64217	13158.400

```
In [1]: import pandas as pd

In [2]: data= pd.read_csv("environment.csv",names=["Temp","Pressure","Humidity","Air Quality"])
        data.head()
```

Out[2]:

	Temp	Pressure	Humidity	Air Quality
0	19.51074	100787.1	61.80923	410444.600
1	19.53587	100785.9	61.80641	10254.170
2	19.53084	100787.7	61.74796	9232.594
3	19.53838	100786.3	61.70413	10364.830
4	19.55095	100793.8	61.66745	11742.580

412 rows at roughly 1 row/second (ignoring the alarm) is just under 7 minutes.

```
In [3]: 412/60
Out[3]: 6.866666666666666
```

A single column can be extracted using its name as a list index:

```
In [20]: airQuality = data['Air Quality']
         airQuality

Out[20]: 0      410444.600
         1       10254.170
         2        9232.594
         3       10364.830
         4       11742.580
                   ...
         407     78621.090
         408     78767.350
         409     78621.090
         410     78621.090
         411     78402.740
         Name: Air Quality, Length: 412, dtype: float64
```

Pandas has a **describe** function which gives us a lot of useful information about our data:

```
In [22]: data.describe()
```

Out[22]:

	Temp	Pressure	Humidity	Air Quality
count	412.000000	412.000000	412.000000	412.000000
mean	19.635302	100783.298058	60.731621	68976.966369
std	0.083940	3.484920	0.373650	23142.179758
min	19.483090	100774.600000	60.182090	9232.594000
25%	19.562893	100780.900000	60.448785	66977.882500
50%	19.632645	100783.300000	60.710455	75738.270000
75%	19.716850	100785.700000	60.838895	77826.330000
max	19.769630	100795.000000	61.809230	410444.600000

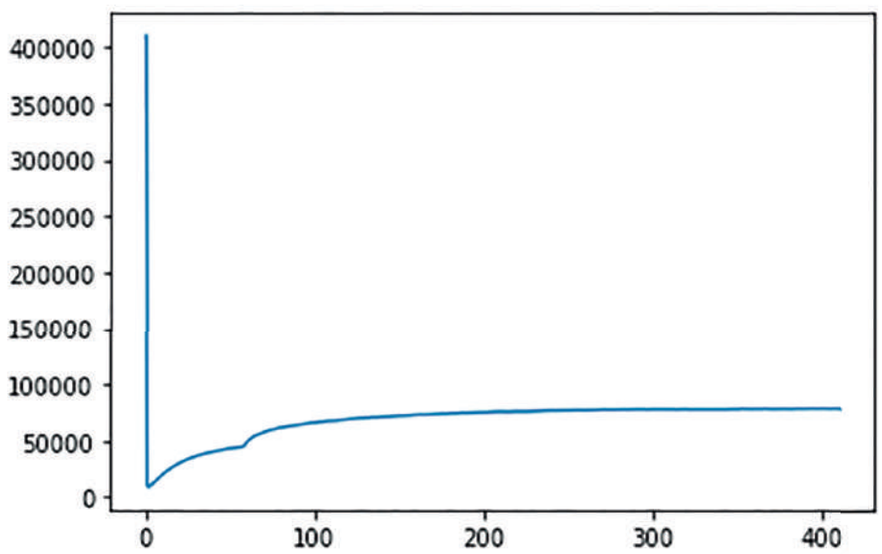
5

Visualizing data is often one of the best ways of seeing what is happening. We can use the library MatPlotLib—a specialist library for data visualization—to plot the data. It first needs importing:

```
In [24]: import matplotlib.pyplot as plt
         plt.close("all")
```

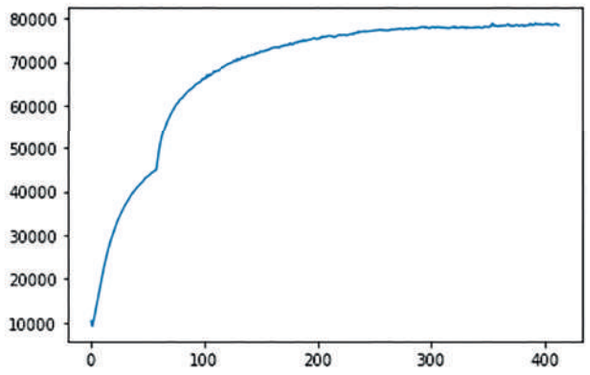
Then the air quality series we extracted earlier can be plotted.

```
In [28]: airQuality.plot()
Out[28]: <matplotlib.axes._subplots.AxesSubplot at 0x7f91edcf63a0>
```



This shows us a substantial problem. The first element was obviously an outlier in this case, and makes the rest of the data largely indistinguishable. The **.iloc** function can be used to "chop off" the first item.

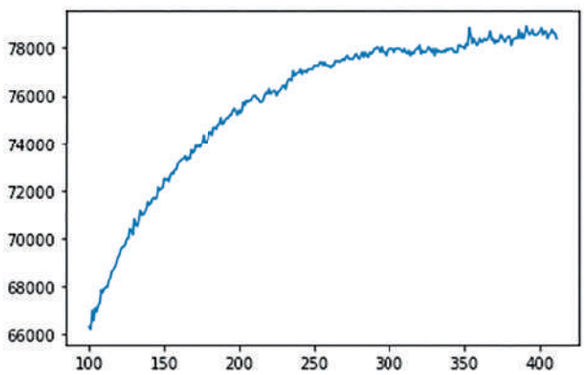
```
In [8]: data = data.iloc[1:]
        data.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 411 entries, 1 to 411
Data columns (total 4 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   Temp         411 non-null    float64
 1   Pressure     411 non-null    float64
 2   Humidity     411 non-null    float64
 3   Air Quality  411 non-null    float64
dtypes: float64(4)
memory usage: 13.0 KB

In [9]: airQuality = data['Air Quality']
        airQuality.plot()

Out[9]: <matplotlib.axes._subplots.AxesSubplot at 0x7fd3a6a4bbe0>
```



And, in fact, we can remove the other 99 first elements to discount the first 100 seconds of "settling in".

The other measurements can be plotted in a similar way; the complete notebook is available in the GitHub repository.

```
In [10]: data = data.iloc[99:]
         data.info()

         airQuality = data['Air Quality']
         airQuality.plot()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 312 entries, 100 to 411
Data columns (total 4 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   Temp         312 non-null    float64
 1   Pressure     312 non-null    float64
 2   Humidity     312 non-null    float64
 3   Air Quality  312 non-null    float64
dtypes: float64(4)
memory usage: 9.9 KB

Out[10]: <matplotlib.axes._subplots.AxesSubplot at 0x7fd3a6c6d2e0>
```



Testing

To fully test this project, you may have to introduce some artificial influences. This could be moving around the room, opening a door or window, or turning on a fan or light.

Stretch tasks

- Add a timestamp so that you can compare when events happened. Plot the timestamp on the OLED as the program is running so you can note any curiosities.
- Increase the time between measurements and run the monitor over a longer period of time. You should be able to store roughly 32,000 samples before you run the risk of running out of memory, so one sample every 15 seconds should give you a good four days' worth of recordings.
- Plot the four sensor readings and see if you can identify causal factors over the four days.
- Explore the use of Pandas and MatPlotLib for plotting data and identify more effective ways you could combine the readings and the timestamps. There is a link in the GitHub repository under Chapter 3 to get you started. You may want to consider grouping data and using a histogram rather than a continuous line graph, for example.

Final thoughts

Data often includes outliers and rogue values. This can be from power surges, as sensors start up (which often takes longer than the microcontroller), due to electrical interference, and a multitude of other things. The jump in the air quality shown actually corresponds to the author standing up and moving away from the desk. If you plot the temperature graph, you will see it starts dropping at a similar time, as does the humidity. You have plotted data over only a limited number of sensors and a relatively short period of time, so you could possibly have got away with using a spreadsheet package instead. Data scientists who monitor the environmental impact of humans on the earth may have thousands of sensors with millions (or even billions) of data points each. Powerful data science tools such as Jupyter Notebooks and the libraries within Python (and similar languages such as R) make manipulating, analyzing, and understanding this vast amount of data far more practicable. As such, they open up opportunities to use it for the greater good in tackling Global Goal 13.

4. THE "I" IN IOT

Setting the scene

No current microcontroller book would be complete these days without at least considering the Internet of Things, or IoT. Using Internet connectivity we can not only monitor our devices from a distance, but also interact with them remotely and feed data into third party systems. Many of the projects we have explored already have looked at environmental data; this chapter is going to do the same, but explore some new techniques which expand the potential of your devices.

In this chapter we will be re-visiting the BME680 sensor, but looking at ways that the monitor could be used over a longer period of time and in remote locations, as well as exploring some alternative interfaces. You might use this to monitor the environment in pollution-heavy areas such as your garage, but equally devices could be put around a school that is concerned with traffic-related pollution to help local government make decisions about road traffic usage and encourage its citizens to use more earth- and child-friendly modes of transport. This could support Global Goal 13 (Climate action), but also Global Goal 7 (Affordable and clean energy) by highlighting causes of pollution.

Success criteria

- ☑ Connect the Pi Pico to the Internet.
- ☑ Create a ThingSpeak endpoint.
- ☑ Collect data from the BME680 and format it for transmission.
- ☑ Transmit data to ThingSpeak and produce a visualization on the web.
- ☑ Use MATLAB to automatically cleanse data and identify outliers.

1

Despite its incredibly low price, the Pico W has built on-board WiFi capabilities, making it ideal for IoT projects allowing access to data stored on the Pico W remotely. As we have already learned, the storage capacity on the Pico W is somewhat limited and adding an SD card to the Explorer board set up will allow us to store much larger data sets. It is worth noting at this point that the Pico W can only handle SD cards with a maximum capacity of 32GB of data.

Now let's connect your Pico W to the internet.

To connect your Pico W to the internet, we need to ensure it can access a website using **HTTP** (Hypertext transfer protocol that manages communications between web servers and clients). While Google.com might seem like an obvious choice, it enforces **HTTPS** (secure HTTP) and tries to redirect any HTTP requests. Since the Pico W cannot natively handle HTTPS connections, using Google will result in an error. Instead, we have commented out Google and focused on a simple HTTP connection.

```
import network  # For WiFi connection
import urequests  # For HTTP requests
import time  # For delays
from machine import Pin # For controlling the onboard LED

# Wi-Fi credentials
SSID = "Your ssid"  # Replace with your WiFi SSID
PASSWORD = "your password"  # Replace with your WiFi Password

# HTTP Settings
HTTP_REQUEST_HOST = "http://google.com"

# Onboard LED setup
led = Pin("LED",Pin.OUT)  # Built-in LED on Pico W

# Connect to WiFi
def connect_to_wifi():
    wifi = network.WLAN(network.STA_IF)
    wifi.active(True)
    led.off()  # LED off while connecting
    print("Connecting to WiFi...")

    wifi.connect(SSID, PASSWORD)

    while not wifi.isconnected():
        time.sleep(0.5)  # Wait until connected
        print("Connecting to WiFi...")

    print("Connected to WiFi!")
    print("IP Address:", wifi.ifconfig()[0])
    led.on()  # LED on when connected

# Perform an HTTP GET request
def make_http_request():
    try:
        response = urequests.get(HTTP_REQUEST_HOST)
        print("HTTP Response Code:", response.status_code)
        print("Response Content:", response.text[:500])  # Show first 100 characters
        response.close()
    except Exception as e:
        print("HTTP Request Failed:", e)

# Main Program
connect_to_wifi()
make_http_request()

```

PRO *TIP*

While so much more powerful than you may expect, microcontrollers are still really quite slow and limited compared to a computer. Python as a norm doesn't really do constants, but using the MicroPython `const()` function will make integers into actual constants, which allows the Pico to save a bit of memory and is generally good practice.

2

Let's explore the code.

We start by importing the necessary libraries: **network** (to handle WiFi connections), **urequests** (for communication over the network), **time** (for delays), and **machine** (for the onboard LED). The WiFi SSID and password are set as constants. A constant is set for the **host**, which specifies the target website (**Google.com**).

The program starts by attempting to connect to WiFi with the provided credentials by calling the `connect_to_wifi()` function. The on-board **LED** serves as a simple status indicator: it remains **off** until a successful Wi-Fi connection is established. Once connected and able to access the website, the **LED turns on**, providing a clear and immediate visual confirmation for students, along with progress messages on the display.

The main program then calls the `make_http_request()` function which makes an HTTP request to our defined host. It then reports the response or if the request fails.

This simplified example focuses on demonstrating **WiFi connectivity**, **basic HTTP requests**, and using the on-board **LED for feedback**. It avoids unnecessary complexity while delivering a clear understanding of how the Pico W can interact with web servers over WiFi.

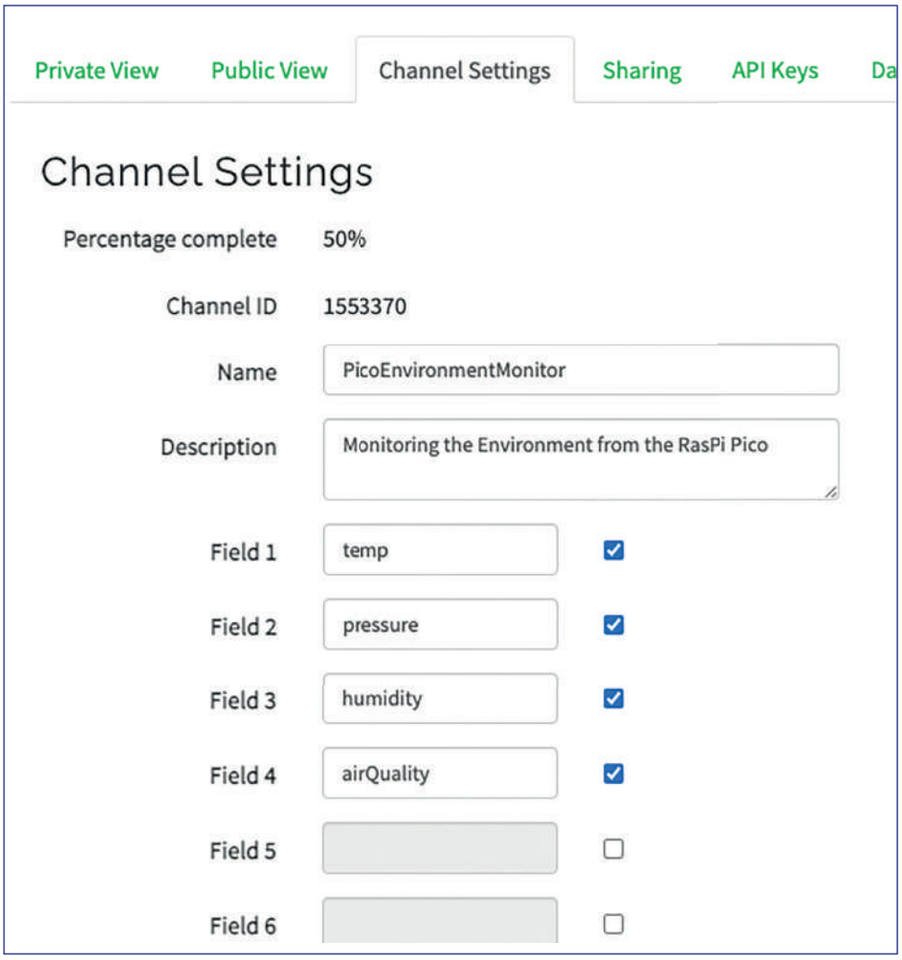
3

ThingSpeak is an open-source software which handily helps users communicate with **Internet-enabled** devices such as the Pico W. It enables users to log, access and retrieve data remotely from the device by providing an **API** (application programming interface—a set of routines that allow one program, in this case your Python code, to interact with another program, in this case, ThingSpeak).

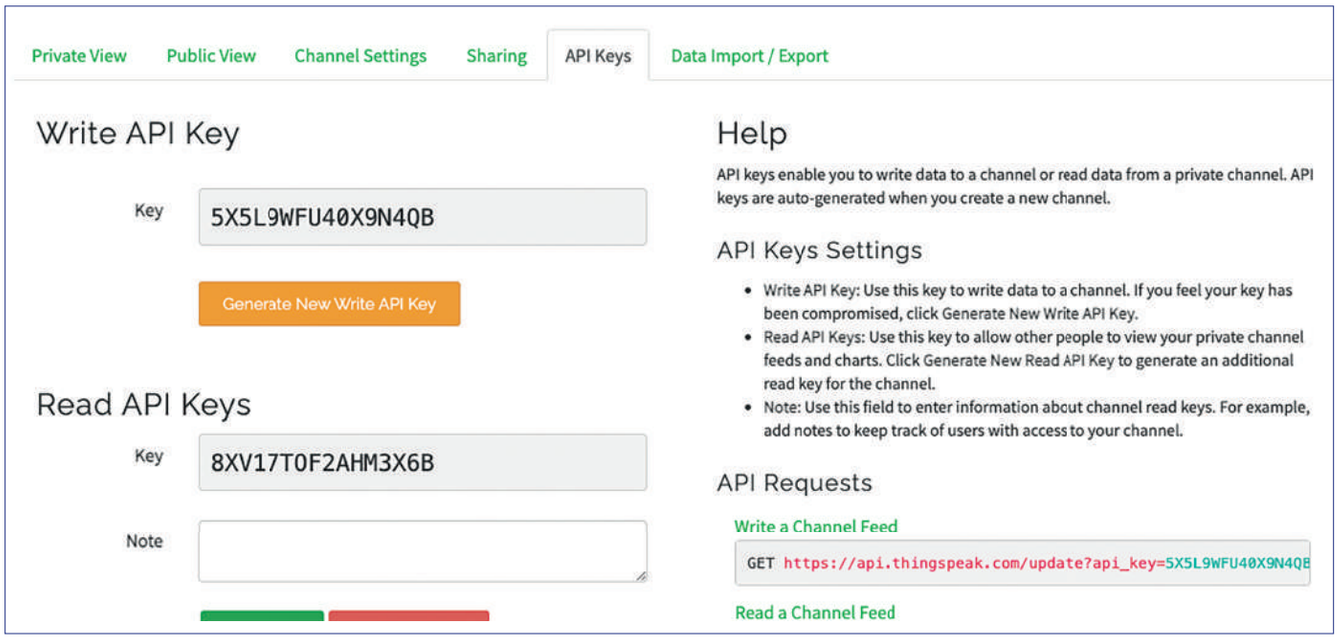
By connecting the Pico W to WiFi, data can be uploaded to the Thingspeak Server using the **API Key** provided with your ThingSpeak account. Once data has been uploaded, you are able to view it in a graphical format.

4

Communication online can be in either direction, either sending instructions to the microcontroller or, in this case, sending data from the microcontroller to somewhere else. We will use a common data science endpoint called ThingSpeak. You can sign up for a free account at https://thingspeak.com. *Endpoints* are generally considered as a unit (device, tool, or service) at the end of a communication channel; in this case your Pico is one endpoint and the ThingSpeak service is the other.



When there you will need to create a channel (data flows through channels) and list the data fields you will store. We went for temperature, humidity, pressure, and air quality. Then check out the API keys and copy the API key as you will use this in your code.



The API keys are the way of identifying which specific instance of the program you wish to interact with. In the case of ThingSpeak, they are a private "identifier" for your channel.

5

Formatting the data is not dissimilar to the method used in the original BME monitoring program. Each value is stored in a variable and formatted appropriately into a string. We captured the API key from the ThingSpeak page earlier; we can add this API to our code and add the floor function import to allow us to round our data later in the code.

```
import network  # For Wi-Fi connection
import urequests  # For HTTP requests
import time  # For delays
from machine import Pin  # For onboard LED control
from pimoroni_i2c import PimoroniI2C
from breakout_bme68x import BreakoutBME68X
from math import floor  # For rounding down

# Wi-Fi Credentials
SSID = "YOUR SSID"
PASSWORD = "YOUR PASSWORD"

# ThingSpeak Settings
THINGSPEAK_API_KEY = "YOUR API"
THINGSPEAK_URL = "http://api.thingspeak.com/update"
```

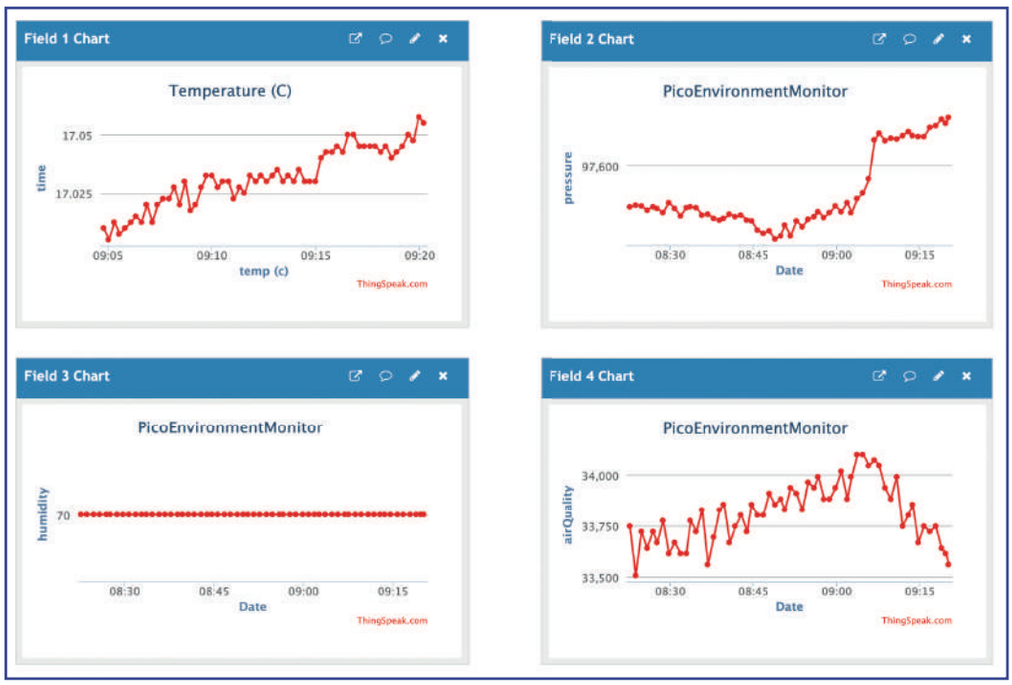
As previously, we need to connect to the WiFi. In this code we have a function that connects to the WiFi and once connected the Pico W on-board LED is turned on. A second function is used to connect to ThingSpeak. Using the API key, this function makes a request to update the Thingspeak channel with the current data from the BME sensor. These functions will be called in the main loop.

```
import time
from breakout_bme68x import BreakoutBME68X
from pimoroni_i2c import PimoroniI2C
from pimoroni import PICO_EXPLORER_I2C_PINS
from picographics import PicoGraphics, DISPLAY_PICO_EXPLORER
import network
import urequests
from math import floor

# Wi-Fi and ThingSpeak Configuration
SSID = "YOUR_WIFI_SSID"
PASSWORD = "YOUR_WIFI_PASSWORD"
THINGSPEAK_API_KEY = "YOUR_API_KEY"
THINGSPEAK_URL = "http://api.thingspeak.com/update"

# Initialize Display
display = PicoGraphics(display=DISPLAY_PICO_EXPLORER)
BLUE = display.create_pen(0, 0, 255)
GREEN = display.create_pen(0, 255, 0)
RED = display.create_pen(255, 0, 0)
WHITE = display.create_pen(255, 255, 255)

# Initialize I2C and BME Sensor
i2c = PimoroniI2C(**PICO_EXPLORER_I2C_PINS)
bme680 = BreakoutBME68X(i2c)

# Initialize Wi-Fi
wlan = network.WLAN(network.STA_IF)
wlan.active(True)
display.set_pen(BLUE)
display.clear()
display.text("Connecting to Wi-Fi...", 10, 10, scale=2)
display.update()

while not wlan.isconnected():
    try:
        wlan.connect(SSID, PASSWORD)
        time.sleep(1)
    except:
        pass

display.set_pen(GREEN)
display.clear()
display.text("Wi-Fi Connected!", 10, 10, scale=2)
display.update()

# Main Loop for Sensor Data and ThingSpeak Upload
while True:
    # Read Sensor Data
    temperature, pressure, humidity, gas, _, _, _ = bme680.read()
    temp = str(temperature)
    pres = str(floor(pressure))
    hum = str(floor(humidity))
    gas = str(floor(gas))

    # Display Data on Explorer Board
    display.set_pen(WHITE)
    display.clear()
    display.text(f"Temp: {temp}C", 10, 20, scale=2)
    display.text(f"Press: {pres}Pa", 10, 40, scale=2)
    display.text(f"Hum: {hum}%", 10, 60, scale=2)
    display.text(f"Gas: {gas}", 10, 80, scale=2)
    display.update()

    # Prepare and Send Data to ThingSpeak
    url = f"{THINGSPEAK_URL}?api_key={THINGSPEAK_API_KEY}&field1={temp}&field2={pres}&field3={hum}&field4={gas}"
    try:
        response = urequests.get(url)
        if response.status_code == 200:
            display.set_pen(GREEN)
            display.text("Upload Success!", 10, 100, scale=2)
        else:
            display.set_pen(RED)
            display.text("Upload Failed!", 10, 100, scale=2)
        display.update()
        response.close()
    except Exception as e:
        display.set_pen(RED)
        display.text(f"Error: {e}", 10, 100, scale=1)
        display.update()

    time.sleep(15)  # Wait before the next upload

```

In this code, the function to connect to wifi is called `connect_to_wifi()`. If this is successful, the sensor is read and the data is stored in variables and converted to strings using the **floor** function to round the data down and finally the call to update the data is made.

6

For every field you include, ThingSpeak automatically creates a visualization. You can edit the axis, styles, etc. of these to improve how your data appears.



It is worth noting that by default only the last 60 readings are shown, which at a reading every 15 seconds is only 15 minutes. We have sample data from 6 p.m. to 9 a.m., so we can calculate 15 hours is 900 minutes, and multiplying this by 4 readings a minute gives us 3,600 readings. The visualizations can be edited to show this.

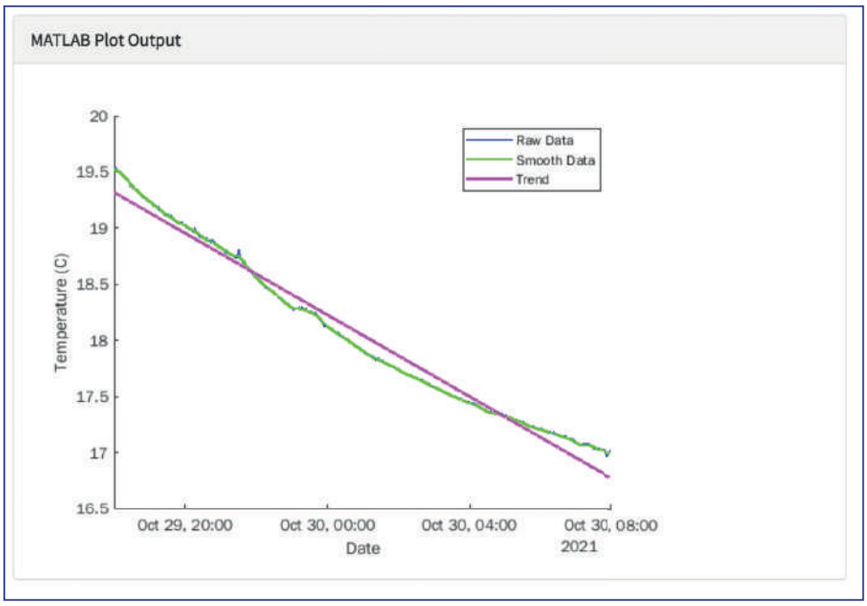
7

ThingSpeak is associated with MATLAB—a programming language which is built into the platform and allows us to do some (if we wish, very complex) data analysis. For now, we will look at the temperature data and how that varied overnight. From your channel you will need to create a MATLAB visualization and choose no starter code.

```
% Channel ID to read data from
readChannelID = 1553370;
% Specify date range
dateRange = [datetime('2021-10-29 18:00:00'),datetime('2021-10-30 08:00:00')];
% Create variables to store different sorts of data
readChannelID = 1553370;
readAPIKey = '8XV17TOF2AHM3X6B';
% Read data including the timestamp, and channel information.
[data,time,channelInfo] = thingSpeakRead(readChannelID,'Fields',1:4,'ReadKey',
readAPIKey,'DateRange',dateRange);

%Read the data into separate variables
temperatureData = data(:,1);
pressureData = data(:,2);
humidityData = data(:,3);
airQualityData = data(:,4);


% Remove missing data from the temperature variable, in order to perform
% the fitting methods
idx = ~isnan(temperatureData);
rawTemp = temperatureData(idx);
newTime = time(idx);

% Smooth the raw temperature data with local 60-point mean values
smoothTemp = movmean(temperatureData(idx),60);

% Fit the data for a trend line
[p,~,mu]= polyfit(datenum(newTime),rawTemp,1);
trend = polyval(p,datenum(newTime),[],mu);


% Plot the raw data, smooth data, trend and fitting curve
figure %Create a plot
hold on %Don't draw it yet
plot(newTime,rawTemp,'b') % Plot the raw temperature
plot(newTime,smoothTemp,'g','LineWidth',1.5) % Plot the smoothing curve
plot(newTime,trend,'m','LineWidth',2) % Plot the trend line
hold off %Draw!
xlabel('Date') %Add the x label
ylabel('Temperature (C)') %Add the y label
legend({'Raw Data','Smooth Data','Trend'},'Location','NE') %Draw the legend
```

You can see the code has a few stark differences to Python! Comments start with a % and lines end with a ; for a start. Let's walk through the code:

- We set up some values—the channel and API keys and the date/times we're interested in (note we shifted back an hour as the data was collected in GMT but is stored as UTC).
- Data is then pulled into **data**, **time**, and **channelInfo** variables, and the data is separated for the four sensor values.
- **isnan** (is not a number) is used to remove any empty values—a little bit of rough data cleansing.
- Smoothed data and trend lines are created.
- Finally, a plot is shown with the raw data, smooth data, and trend lines.



You can see our data was already pretty smooth as the sensor was in a closed room overnight. The plot will change depending on time periods and where the sensor is.

Testing

This project takes some time to test fully. It is worth leaving the sensor for a good period of time—possibly even a full 24 hours. It is also worth experimenting with different areas. For example, closed doors, room occupancy, and proximity to a shower will all affect the variations you will see, and the more test data you have the more you can learn.

Stretch tasks

- Add MATLAB analysis visualizations for the other sensors and compare the variations.
- Collect data from different locations: the kitchen, near the bathroom, near a doorway, and in a quiet room. These would all give different changes. Write up your conclusions from the plots.
- Explore the examples from MATLAB—particularly the weather station—consider what other data analysis you could use within your own projects.

Final thoughts

Variations in sensor readings can be incredibly useful. Spikes (up or down) in temperatures could indicate times when either the heating or the air conditioning is turning on—both of which use huge amounts of gas or electricity. Could you educate your family on the benefits of keeping doors closed (or windows open)? You could add a buzzer to encourage you to open a window rather than letting the air conditioning turn on automatically. Air conditioning—just in terms of power usage (and ignoring the potentially damaging coolant)—has a huge impact on the environment. During a recent heatwave, 50% of the power usage in Beijing was running air conditioning units. Saving power, whether heating or cooling, can significantly reduce the use of fossil fuels and the environmental impact of generating electricity, working toward developing Global Goal 13 (Climate action) and encouraging Global Goal 7 (Clean energy).

5. DATA SCIENCE FOR MANAGING WELL-BEING

Setting the scene

Social media is a fun, and common, way to pass time, keep in touch with friends, and entertain yourself. Overconsumption, however, can have a negative effect on mental health. It can trigger the hormones that alleviate stress and make you feel happier, but too much time on social media can worsen anxiety and depression. As some teenagers spend on average nine hours a day online, with the most popular social media site being YouTube, their health and well-being may be at risk, something that we all need to work on to achieve Global Goal 3, Good health and well-being.

In this chapter you will be introduced to a number of new techniques, including connecting up a *potentiometer* (a variable resistor) and mapping the values. The data you capture will model your usage of social media over the course of a day, which you will analyze in Jupyter Notebook and, hopefully, be able to use to keep an eye on some aspects of your own well-being.

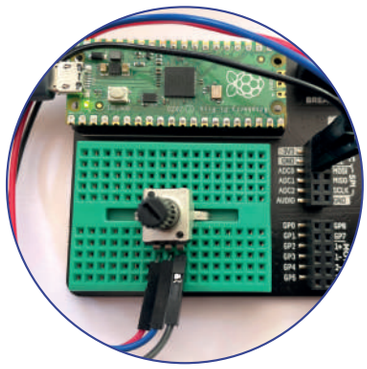
Success criteria

- ☑ Use the Pi Pico to capture button presses and analog data.
- ☑ Map analog data to useful values.
- ☑ Map the concept of a finite state machine to a program.
- ☑ Collate, store, and transfer data to a notebook in Jupyter.
- ☑ Graph, sort, and search data and draw conclusions.

1

This project will need to capture two kinds of data: digital data from the button presses to indicate which social media site you have been using, and analog data so that you can dial in how **much** time you have been on them for. We explored capturing button presses in an earlier chapter, but handling analog data directly is new to us (although the sensors we have used previously report analog data, this has been abstracted by the libraries that handle them). We will use a potentiometer—or pot for short—which has a positive voltage (red) and a ground (black) connected to either end, and a signal wire (blue) (normally from the center terminal) which varies depending on how far around the potentiometer is turned.

Digital data, or discrete values, are all that computers really understand. *Binary* is the ultimate digital data, as it can be in only two states, but any data which "jumps" between values is discrete and therefore can be represented directly in digital format. *Analog data* is continuous. The temperature may start off at 20 degrees and then move to 21 degrees, but it has to go via every infinitesimally small value on its way—there is no way it can jump. This continuous data is a bit tricky for computers—an infinite number of "steps" cannot be recorded, so the computer has to use an analog-to-digital converter (ADC) to approximate it as closely as possible.



Note we have left the audio jumper connected as we will be using this later in the project. The code below will read the ADC0 pin which the potentiometer is connected. This is actually connected to GPIO 26 on the Pico, and again this can be seen on the underside of the Pico Explorer board. This comes as a floating point number, so dividing by 100 and converting it to an integer gives us something a bit easier to use. It will then output the reading to the screen.

```
from machine import Pin, ADC
from time import sleep  # For delays


# Set up potentiometer on ADC pin 26
pot = ADC(Pin(26))


# Main Loop
while True:
    # Read potentiometer value (0-65535), scale it to 0-100
    pot_value = pot.read_u16()
    mapped_value = int(pot_value /100)
    print(f"Potentiometer: {pot_value} -> Mapped: {mapped_value}")

    sleep(0.1)  # Pause for 0.1 seconds before repeating

```

We can extend this to also check whether a button has been pressed:

```
from machine import Pin, ADC
from time import sleep  # For delays
from pimoroni import Button

# Set up buttons
button_a = Button(12)
button_b = Button(13)
button_x = Button(14)
button_y = Button(15)

# Set up potentiometer on ADC pin 26
pot = ADC(Pin(26))

# Map a value from one range to another
def map_value(value, in_min, in_max, out_min, out_max):
    return int((value - in_min) * (out_max - out_min) / (in_max - in_min) + out_min)


# Main Loop
while True:
    # Read potentiometer value (0-65535), scale it to 0-100
    pot_value = pot.read_u16()
    mapped_value = map_value(pot_value, 0, 65535, 0, 100)
    print(f"Potentiometer: {pot_value} -> Mapped: {mapped_value}")

    # Button Inputs
    if button_a.is_pressed:
        print("Button A Pressed")
    if button_b.is_pressed:
        print("Button B Pressed")
    if button_x.is_pressed:
        print("Button X Pressed")
    if button_y.is_pressed:
        print("Button Y Pressed")

    sleep(0.1)  # Pause for 0.1 seconds before repeating
```

PRO *TIP*

Potentiometers do vary. We have used a 10k potentiometer from an electronics set. Make sure you check yours is wired up the same way. For this (and any other technical details you may need) find the model number on the component and type it into your favorite search engine. You should find a datasheet which holds all of the technical details.

2

We want to be able to use the potentiometer to dial in the number of minutes spent on social media in a 15-minute period. If you have ever used an Arduino, you may be aware of the excellent **map** function which will convert a value to be within any range we need. Although there exists a **map** function in Python, it does something slightly different. However, we can write our own **map** function (based on the Arduino code in fact) which takes the input value and the min and max of the original and new ranges, and returns an integer value within the new range. We can then use this to plot an appropriate time period on the screen.

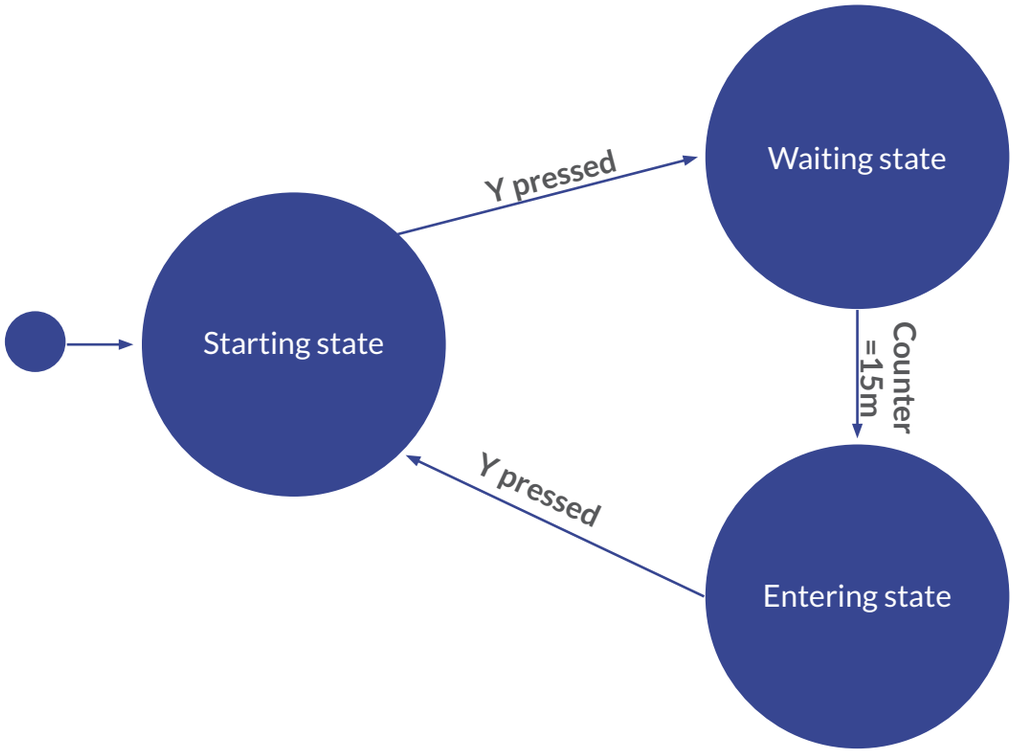
PRO *TIP*

Quite often, when a function exists in one language but not another, you can delve into the source code of one language and re-implement it in another. In this case we renamed map to map_value so as not to interfere with the built-in Python map function.

```
from machine import Pin, ADC
from time import sleep  # For delays
from pimoroni import Button

# Set up buttons
button_a = Button(12)
button_b = Button(13)
button_x = Button(14)
button_y = Button(15)

# Set up potentiometer on ADC pin 26
pot = ADC(Pin(26))

# Map a value from one range to another
def map_value(value, in_min, in_max, out_min, out_max):
    return int((value - in_min) * (out_max - out_min) / (in_max - in_min) + out_min)


# Main Loop
while True:
    # Read potentiometer value (0-65535), scale it to 0-100
    pot_value = pot.read_u16()
    mapped_value = map_value(pot_value, 0, 65535, 0, 100)
    print(f"Potentiometer: {pot_value} -> Mapped: {mapped_value}")

    # Button Inputs
    if button_a.is_pressed:
        print("Button A Pressed")
    if button_b.is_pressed:
        print("Button B Pressed")
    if button_x.is_pressed:
        print("Button X Pressed")
    if button_y.is_pressed:
        print("Button Y Pressed")

    sleep(0.1)  # Pause for 0.1 seconds before repeating

```

3

You may already be aware of a finite state machine from your other learning. If you are not, it refers to modeling a machine that can be in one and only one state at a time and is able to move between these states based on occurrences within the running of the machine. We can replicate this model in our program. This program will have a starting state a waiting state (while it waits for a 15-minute period to pass), and an entering state where you enter the social media usage for the last 15 minutes. We will have buttons A, B, and X used for identifying three main social media sites and the Y button to confirm. This model can be seen here:



We can model this in Python with a **state** variable and use selection statements to execute the relevant code, as shown in this code template:

```
#STATES
START = 0
WAITING = 1
ENTERING = 2

state = START

while True:
    if state == START:
        #initialise anything needed
        #IF button pressed (Y)
        #THEN state = WAITING
    elif state == WAITING:
        #Increment time counter
        #Wait 1 second
        #IF time counter is 15 minutes
        #THEN state = ENTERING
    elif state == ENTERING:
        #Capture button presses
        #Capture time spent
        #Save to file
        #IF button pressed (Y)
        #THEN save to file and state = WAITING

```

4

Now we have a template, we can put all of our code in the relevant state sections:

- **START** initialize variables
- **WAITING** count for 15 minutes
- **ENTERING** capture button presses/time and save to file

```
from pimoroni import Button
from picographics import PicoGraphics, DISPLAY_PICO_EXPLORER
from machine import ADC, Pin
from time import sleep

# Set up the display
display = PicoGraphics(display=DISPLAY_PICO_EXPLORER)
WIDTH, HEIGHT = display.get_bounds()

# Set up audio pin
audio_pin = Pin(0, Pin.OUT)

# Buttons
button_a = Button(12)
button_b = Button(13)
button_x = Button(14)
button_y = Button(15)

# Potentiometer
pot = ADC(Pin(26))

# Map a value from one range to another
def map_value(value, in_min, in_max, out_min, out_max):
    return int((value - in_min) * (out_max - out_min) / (in_max - in_min) + out_min)

# File
FILENAME = "socialmedia.csv"

# STATES
START = 0
WAITING = 1
ENTERING = 2

period = 0  # Incrementing period to keep track of recordings
MAXCOUNTER = 60 * 15  # 15 minutes
state = START  # Start state

# Social Media State
social_media = [['YouTube', False, 20, 20],
                ['Facebook', False, 20, 200],
                ['Pinterest', False, 140, 20]]

def display_text(text, x, y, color=(0, 0, 0), size=2):
    display.set_pen(display.create_pen(*color))
    display.text(text, x, y, WIDTH, size)
    display.update()

# Main Loop
while True:
    if state == START:
        counter = 0
        display.set_pen(display.create_pen(255, 255, 255))
        display.clear()
        display_text('Press Y to start', 60, 100)
```

```
55
56          if button_y.is_pressed:
57              state = WAITING
58
59          with open(FILENAME, "w") as f:
60              f.write('period,YouTube,Facebook,Pinterest,time_spent\n')
61
62          sleep(0.2)
63
64      elif state == WAITING:
65          display.set_pen(display.create_pen(255, 255, 255))
66          display.clear()
67          display_text('Waiting...', 60, 100)
68          display_text(f"{MAXCOUNTER - counter} s", 60, 120)
69          counter += 1
70          sleep(1)
71
72          if counter == MAXCOUNTER:
73              counter = 0
74              state = ENTERING
75
76      elif state == ENTERING:
77          display.set_pen(display.create_pen(255, 255, 255))
78          display.clear()
79
80          for i, sm in enumerate(social_media):
81              color = (0, 255, 0) if sm[1] else (255, 0, 0)
82              display_text(sm[0], sm[2], sm[3], color)
83
84          # Button Controls
85          if button_a.is_pressed:
86              social_media[0][1] = not social_media[0][1]
87          if button_b.is_pressed:
88              social_media[1][1] = not social_media[1][1]
89          if button_x.is_pressed:
90              social_media[2][1] = not social_media[2][1]
91
92          sleep(0.2)
93
94          # Potentiometer Input
95          pot_value = pot.read_u16()
96          pot_mapped = map_value(pot_value, 0, 65535, 0, 15)
97          display_text(f"{pot_mapped} minutes", 60, 150, (0, 0, 255))
98
99          # Confirmation
100         display_text("Confirm (Y)", 140, 200)
101         if button_y.is_pressed:
102             period += 1
103             with open(FILENAME, "a") as f:
104                 f.write(f"{period},{social_media[0][1]},{social_media[1][1]},{social_media[2][1]},{pot_mapped}\n")
105             state = WAITING
106
107     display.update()
108     print(state)
```

You can see in our code we have done a few user experience things, such as color coding whether each type of social media has been selected or not and using lists to organize data a bit more effectively. Once you have collected enough data, transfer it to your computer to analyze in Jupyter Notebook.

PRO *TIP*

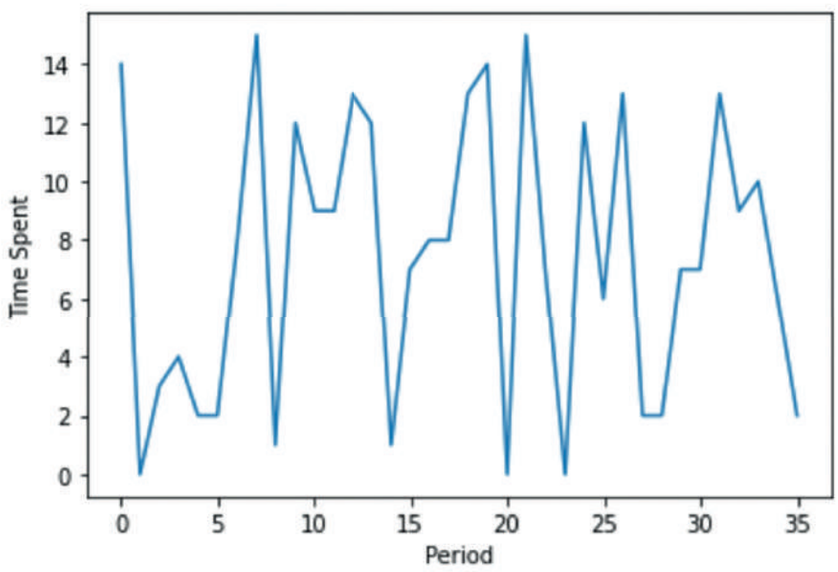
If your Raspberry Pi Pico freezes up during testing, we found unplugging the larger USB plug from our USB hub was easier (and less prone to damage) than unplugging the micro USB from the Pico itself. This speeds up troubleshooting.

5

When you describe the data in Jupyter Notebook, you will see only period and time spent are listed, as this is the only data that is numerical (and therefore can have numerical statistics). On our data, a plot of **time_spent** showed that we were quite inconsistent with how much time we spent on social media, although with a mean of 7.4 minutes in a 15-minute period we obviously didn't get much work done!

```
In [5]: plt.xlabel("Period")
        plt.ylabel("Time Spent")
        data['time_spent'].plot()
```

```
Out[5]: <matplotlib.axes._subplots.AxesSubplot at 0x7f89a87c7d60>
```



Using `data.column_name.value_counts()` shows us how many times we did, or did not, use a particular social media platform. We used YouTube the most, but the three platforms were quite close.

```
In [12]: data.YouTube.value_counts()

Out[12]: True     21
         False    15
         Name: YouTube, dtype: int64

In [13]: data.Facebook.value_counts()

Out[13]: True     20
         False    16
         Name: Facebook, dtype: int64

In [14]: data.Pinterest.value_counts()

Out[14]: False    19
         True     17
         Name: Pinterest, dtype: int64
```

To see how each of the three compares to how much time we spent on social media, we first need to add a total to see how many platforms we were using.

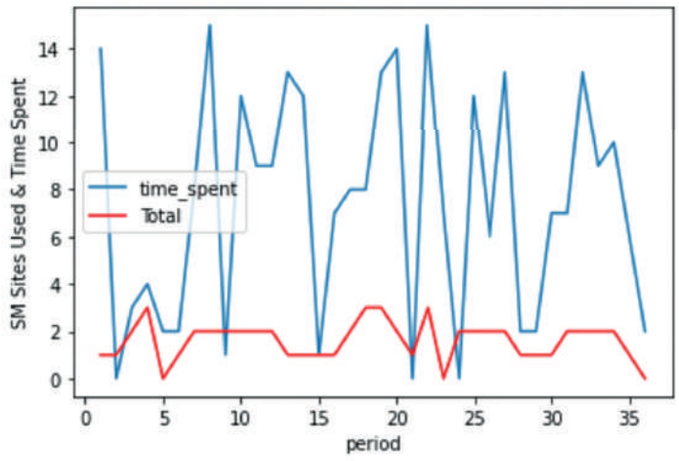
```
In [9]: data['Total']=data.select_dtypes(include=['bool']).sum(axis=1)

In [10]: data
```

Out[10]:

	period	YouTube	Facebook	Pinterest	time_spent	Total
0	1	True	False	False	14	1
1	2	False	False	True	0	1
2	3	False	True	True	3	2
3	4	True	True	True	4	3
4	5	False	False	False	2	0
5	6	True	False	False	2	1
6	7	True	False	True	8	2

We can then plot them together to look for some correlations:

```
ax = plt.gca()

plt.xlabel("Period")
plt.ylabel("SM Sites Used & Time Spent")
data.plot(kind='line',x='period',y='time_spent',ax=ax)
data.plot(kind='line',x='period',y='Total', color='red', ax=ax)

plt.show()
```



So, it does look as if the number of social media platforms we were on increases with the amount of time we spent on social media. We can also create a new DataFrame (the data structure that holds the data—a bit like a spreadsheet table) based on certain criteria from the existing DataFrame. For example, when we select only data where the time spent on social media is more than 13 (so pretty much all of the time we had), we see:

```
In [13]: high_usage = data[data['time_spent']>13]
         high_usage
```

Out[13]:

	period	YouTube	Facebook	Pinterest	time_spent	Total
0	1	True	False	False	14	1
7	8	True	False	True	15	2
19	20	False	True	True	14	2
21	22	True	True	True	15	3

So, what conclusions can we draw from our data? Well, we spend too much time on social media! We searched our data for high-usage periods and found there is no particular platform which is eating all of our time, so we're changing between the three quite a lot.

When looking at data, it is important to either form a hypothesis and test it (we could have believed we spend too much time—where too much is more than a quarter—on social media, or that we spend most of our time on YouTube), or to look for correlations or lack of correlations. What is really important is to not try and force conclusions where there is insufficient evidence.

Testing

When testing, a 15-minute wait time is quite a lot. You can reduce the time down to a few seconds between moving states while making sure everything works. Also test different delay times between detecting button presses to find out what time reliably detects single presses.

Stretch tasks

- It is very easy to miss a "check time", so make the buzzer sound every 15 minutes to remind you to enter the social media usage.
- During the setup state, allow the user to rotate the potentiometer to scroll through different social media platforms, selecting them with the Y button. Make sure you enter more than three social media platforms this time.
- Adjust the entering state so the user enters how long they spent on each platform.
- Add a second entering state and allow users to store their feelings (happy, unhappy, or in between) at each entry point.
- Analyze your own data and draw your own conclusions. They will probably be a lot more exciting than ours.

Final thoughts

Our test data was completed on a day off, which explains the incredibly high usage of social media. While this is enjoyable, regular overuse could potentially have deep social and psychological impacts and have a negative impact upon your mental well-being. Pay careful attention to the conclusions you are able to form and consider whether you may need to make adjustments so that you can look after your health while still enjoying the use of social media, doing your own bit to meet Global Goal 3.

Another interesting point can come out of this project, and that is one of e-waste. In 2019 it was estimated that over 50 million metric tonnes of e-waste was produced worldwide. Although our potentiometer came from a kit, you can often find such things in old toys and radios—a bit of creative de-soldering can produce an array of reusable components. Do be careful though, and don't touch anything you don't understand; even a small capacitor can store a charge and give you a shock!

6. ACCESSING DATA REMOTELY

Setting the scene

Many areas suffer from water shortages, not just in extreme climates but even more moderate ones. The need for clean, drinkable water is universal and a Global Goal. Hosepipe bans and water shortages are common across England in the summer months. Lake Mead hit all-time lows in 2021, leading to water shortages across the western United States.

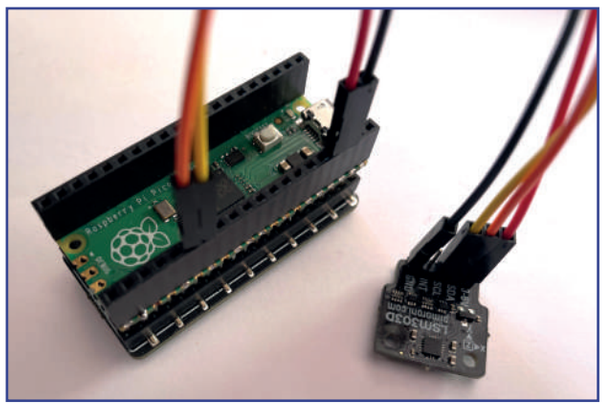
One of the worst culprits of water waste resides in the smallest room in the house—the humble lavatory can flush away as much as 35 gallons a day **per person**—and that's not including the amount that is wasted by leaky lavs—the 5–8% of toilets that leak waste another 100 gallons a day **each**. In this project we're going to use an accelerometer to monitor flushes and turn our devices into IoT devices to enable us to access data remotely. Knowledge is power, and supporting more sustainable water usage directly supports Global Goal 6 in the sustainable management of water and sanitation.

Success criteria

- ☑ Connect the accelerometer and read it through I2C.
- ☑ Calculate the averages of the accelerometer over time.
- ☑ Store data on the SD card via SPI.
- ☑ Serve the data over HTTP and read it with Jupyter Notebook.
- ☑ Analyze the data in Jupyter Notebook to identify trends.

1

In this chapter we will make use of the ICM20948, a movement sensor which can detect motion in three different axis: acceleration, compass heading and gyroscopic. It comes on a handy Breakout Garden format for the Pico Explorer board, allowing us to simply connect it to the 2nd breakout slot on the Pico Explorer board. The sensor address will be different from the BME680 sensor so we can distinguish



between the data from the BME680 even though they are on the same channel. You can find the address of your sensor in the sensor's datasheet or by running the `I2C.scan()` function (the sample code **scanI2c.py** will do this). In this case, the default address of the sensor is 0x68.

Unfortunately, the Pimoroni version of the library for the sensor is not supported on the Pico at the time of writing. As such, we will make a simple class to allow us to simplify the use of the sensor.

```
import time
import struct
from machine import I2C

class ICM20948:
    def __init__(self, i2c, addr=0x68):
        self.i2c = i2c
        self.addr = addr
        self._bank = -1

        # Check WHO_AM_I register
        if self._read_byte(0x00) != 0xEA:
            raise RuntimeError("ICM20948 not found at address 0x68")

        # Reset and initialize
        self._write_byte(0x06, 0x80)  # Reset
        time.sleep(0.1)
        self._write_byte(0x06, 0x01)  # Set clock
        self._write_byte(0x07, 0x00)  # Enable all axes

    def _read_byte(self, reg):
        return self.i2c.readfrom_mem(self.addr, reg, 1)[0]

    def _write_byte(self, reg, value):
        self.i2c.writeto_mem(self.addr, reg, bytes([value]))

    def read_accelerometer(self):
        data = self.i2c.readfrom_mem(self.addr, 0x2D, 6)
        x, y, z = struct.unpack(">hhh", data)
        return x / 16384, y / 16384, z / 16384

    def read_gyroscope(self):
        data = self.i2c.readfrom_mem(self.addr, 0x33, 6)
        x, y, z = struct.unpack(">hhh", data)
        return x / 131, y / 131, z / 131
```

Explanation of the code:

Initialization (__init__): The sensor's I2C address is set (default: 0x68), and the class verifies communication by checking the **WHO_AM_I** register for the expected value (0xEA). The sensor is then reset and configured to enable all axes of the accelerometer and gyroscope.

Next four private methods are defined:

- **read_byte:** Reads a single byte from a specified register on the sensor.
- **write_byte:** Writes a single byte to a specified register.
- **read_accelerometer:** Reads 6 bytes of raw data (*x*-, *y*-, *z*-axes) from the accelerometer's output registers. It then converts the raw data to g units by dividing by the full-scale range factor (default: ±2g → scale = 16384).
- **read_gyroscope:** Reads 6 bytes of raw data (*x*-, *y*-, *z*-axes) from the gyroscope's output registers. It then converts the raw data to degrees per second (dps) using the default scale factor (±250dps → scale = 131).

This code should be saved to the Pico W as **icm20948.py**. We will then import this into our main programs.

We are now ready to test our class and the sensor.

```
from machine import I2C, Pin
from time import sleep
from icm20948 import ICM20948

# Initialize I2C and sensor
i2c = I2C(0, scl=Pin(21), sda=Pin(20))
imu = ICM20948(i2c)

while True:
    ax, ay, az = imu.read_accelerometer()
    gx, gy, gz = imu.read_gyroscope()

    print(f"Accel: X={ax:.2f}, Y={ay:.2f}, Z={az:.2f}")
    print(f"Gyro: X={gx:.2f}, Y={gy:.2f}, Z={gz:.2f}")

    sleep(0.5)
```

Explanation of the code:

From **machine** we import **I2C** and **Pin**, **sleep** is imported from **time**, and we then import the **ICM20948** class object we have just created. We initialize the **i2c** object and then the sensor as **imu**. In the main code, we read in the *x*, *y* and *z* data for the accelerometer and the gyroscope. These are then output to the console.

PRO *TIP*

The ICM20948 is really sensitive, so you will see a bit of movement on the *x*- and *z*-readings all of the time. More importantly, you will see a constant reading of 9.8 metres per second per second on the *y*-axis (if you have the sensor oriented properly). This is because of gravity. You might like to explore the adjustments you can make to the ICM20948 to alter the sensitivity—these can be found in the datasheet.

2

To move away from the background acceleration, we will record only changes in acceleration. We can do this by storing the previous reading in an "old" variable, calculating the difference, and then moving the "new" readings into the "old" variables ready for next time. Of course, you have to ensure you skip this process for the first reading as you have no "old" readings to use. This is what is known as an edge case—something that only occurs at the extremes of the operating parameter (in this case, only on the first reading). At the same time, with 10 readings a second there is very quickly a need to be able to tell when a change happened. For this, we will count up in minutes, seconds, and tenths of seconds to

give a useful time point. If you are taking readings over extended periods, you could of course use the same method to count hours or even days. Note how the gyroscope has been removed in this code as it is not needed.

```
from machine import I2C, Pin
from time import sleep, ticks_ms, ticks_diff
from icm20948 import ICM20948



# Initialize I2C and sensor
i2c = I2C(0, scl=Pin(21), sda=Pin(20))
imu = ICM20948(i2c)

# Variables for tracking changes
old_x, old_y, old_z = None, None, None  # Initialize previous readings
time_start = ticks_ms()  # Start time for timestamp calculation

# Main Loop
while True:
    # Read current accelerometer data
    new_x, new_y, new_z = imu.read_accelerometer()

    # Calculate timestamp in seconds
    elapsed_time = ticks_diff(ticks_ms(), time_start) / 1000.0

    # Skip the comparison for the first reading (edge case)
    if old_x is None:
        old_x, old_y, old_z = new_x, new_y, new_z
        print(f"Initial Reading @ {elapsed_time:.2f}s: X={new_x:.2f}, Y={new_y:.2f}, Z={new_z:.2f}")
    else:
        # Calculate differences
        diff_x = abs(new_x - old_x)
        diff_y = abs(new_y - old_y)
        diff_z = abs(new_z - old_z)

        # Only log significant changes
        threshold = 0.1  # Adjust as needed to filter noise
        if diff_x > threshold or diff_y > threshold or diff_z > threshold:
            print(f"Change @ {elapsed_time:.2f}s: X={new_x:.2f}, Y={new_y:.2f}, Z={new_z:.2f}")

            # Update old readings
            old_x, old_y, old_z = new_x, new_y, new_z

    sleep(0.1)  # 10 readings per second

```

3

The amount of data being stored is quite significant; with ten samples a second, the Pico would run out of space in about an hour. It makes sense therefore to make use of an SD card adapter that can very easily be connected to the Pico via SPI, which stands for Serial Peripheral Interface. There is an **sdcard.py** file (which is available from the GitHub repository) that handles the low-level access for you. You will also need the **uos** library (included with your Pico firmware) to access the SD file system.

```
# SD Card Setup
sck = Pin(18)
mosi = Pin(19)
miso = Pin(16)
cs = Pin(5)
spi = SPI(0, sck=sck, mosi=mosi, miso=miso)
sd = SDCard(spi, cs)
uos.mount(sd, "/sd")
```

The SD card is set up with four pins, **sck** (serial clock for synchronization), **mosi** (the master line for sending data to the SD card), **miso** (for sending data back to the Pico) and **cs** (for selecting and awakening the device).

Once the card is initialized, it can be mounted to the mountpoint/SD using **uos**, and then accessed just like any other text file in Python. We used the **with** style to write a heading to the file (creating it in the process), and then adjusted the loop to append the timepoint and the differences. The code below shows how to write the headers to the file. We have added some error detection as the SD cards may produce errors. Sometimes a soft reboot is needed when this happens.

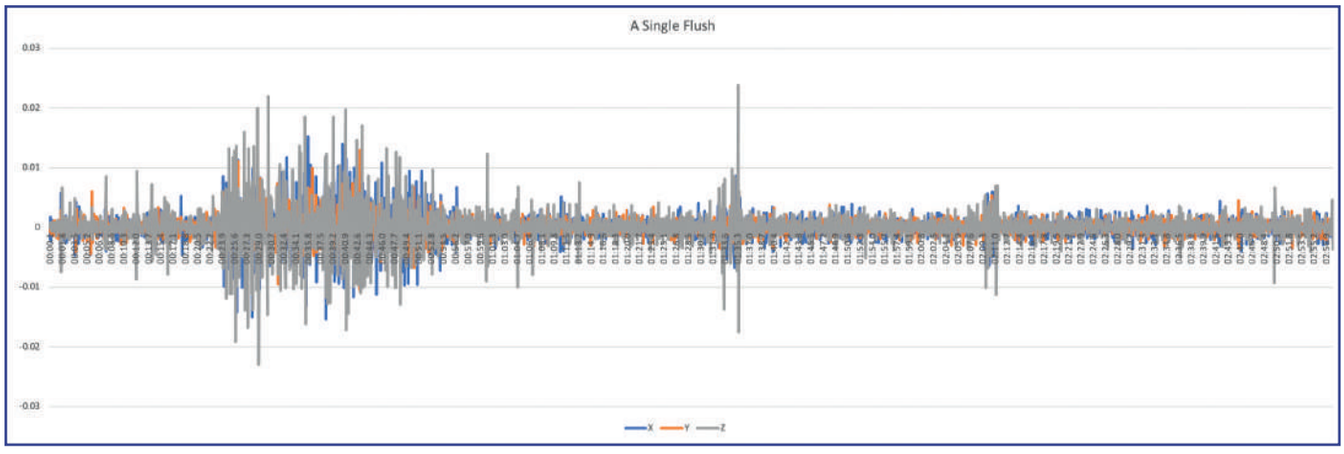
PRO *TIP*

Our method is to create a new data file every time the Pico turns on. This of course destroys the old file, so if you want to keep your data it might be worth creating an algorithm to back up or rename the existing file first. Of course, you could open in append mode, but this would make it harder to differentiate between readings.

```
18  # Initialize SD Card
19  try:
20      print("Initializing SD card...")
21      sd = SDCard(spi, cs)
22      uos.mount(sd, "/sd")
23      print("SD card mounted successfully!")
24
25      # Create a new file and write the header
26      FN = "/sd/accel_data.csv"
27      with open(FN, "w") as f:
28          f.write("Time,X,Y,Z\n")
29  except OSError as e:
30      print(f"SD card error: {e}")
31      raise SystemExit()
```

4

Now that you have a working sensor, it is time to test it. We collected a small sample by taping the sensor to the inlet pipe of the downstairs toilet, letting the sensor settle, then flushing. We disconnected it once the cistern had stopped refilling and put the SD card into our computer to access the data file. Opening the file in a spreadsheet lets us quickly check the data and create a line graph.



Having to unplug the SD card every time you want to look at the data is somewhat annoying, however, and does not help us with remote access. Instead, we want to be able to access the data remotely. Starting an HTTP server on the Pico W is fairly easy; we already have the wireless capabilities and the libraries available. We can make use of the code we previously used to connect to WiFi and then create a socket with `server = start_server()`.

```
# Wi-Fi Setup
def connect_to_wifi():
    wlan = network.WLAN(network.STA_IF)
    wlan.active(True)
    wlan.connect(SSID, PASSWORD)
    print("Connecting to Wi-Fi...")
    while not wlan.isconnected():
        time.sleep(1)
    print("Connected to Wi-Fi!")
    print("IP Address:", wlan.ifconfig()[0])

connect_to_wifi()

# HTTP Server Setup
def start_server():
    addr = socket.getaddrinfo('0.0.0.0', 80)[0][-1]
    s = socket.socket()
    s.bind(addr)
    s.listen(1)
    print("Listening on", addr)
    return s

server = start_server()

# Variables for tracking changes
old_x, old_y, old_z = None, None, None
start_time = time.ticks_ms()

print("Logging accelerometer data and serving it over HTTP...")
```

Things then get a little more complex! The Pico WiFi library has a handy "routes" system which lets you respond with a bit of HTML code very simply. However, we need to send a quite large amount of data and therefore run the risk of (a) having memory issues and (b) timing out. Instead, we need to create a separate function to handle file requests (and totally ignore anything else).

```
        # Serve data over HTTP
        client, addr = server.accept()
        print("Client connected from", addr)
        request = client.recv(1024)
        request = str(request)
        if "/data" in request:
            with open(FN, "r") as f:
                response = "HTTP/1.1 200 OK\r\nContent-Type: text/plain\r\n\r\n"
                response += f.read()
            client.send(response)
        else:
            client.send("HTTP/1.1 404 Not Found\r\n\r\n")
        client.close()

        time.sleep(0.1)  # Log 10 readings per second

    except Exception as e:
        print(f"Error: {e}")
        break
```

Let's talk through the code.

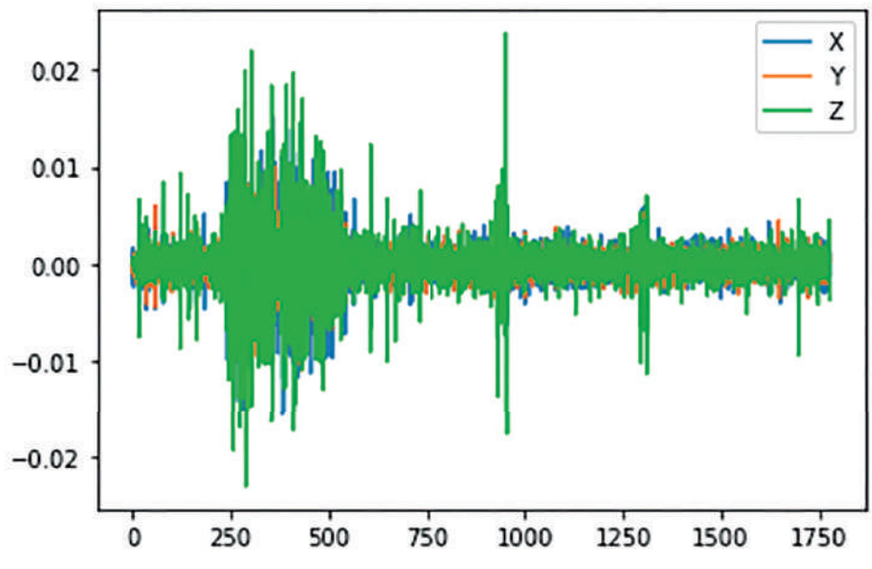
- A client socket is created and verified.
- The request itself is checked; this provides a lot of useful information, although we just want to check if the /data url is being requested so we don't worry about anything else.
- The correct HTTP header, code 200, is sent.
- At this point, as the data is small, we can just send it all.
- Any requests other than /data get a 404 not found error.

You can now create a notebook in Jupyter on your computer (as long as it is on the same WiFi network as your Pico) and load in the CSV file using `data= pd.read_csv("http://192.168.1.157/data.csv") #Get the live data`—where the IP address shown is that of your Pico, of course.

5

Identifying the flushes in Jupyter Notebook is relatively simple, although you may need to use a bit of trial and error. We used three libraries: Pandas and MatPlotLib, as we've used before, and NumPy (imported as **np**). NumPy is a package for scientific computing; it provides many routines for all sorts of scientific mathematics—we will use it for some list DataFrame modification. It may also be worth saving a "good" set of data (or using ours) for testing purposes to save pulling data from the Pico all of the time.

```
Out[7]: <matplotlib.axes._subplots.AxesSubplot at 0x7ff779f4d6a0>
```

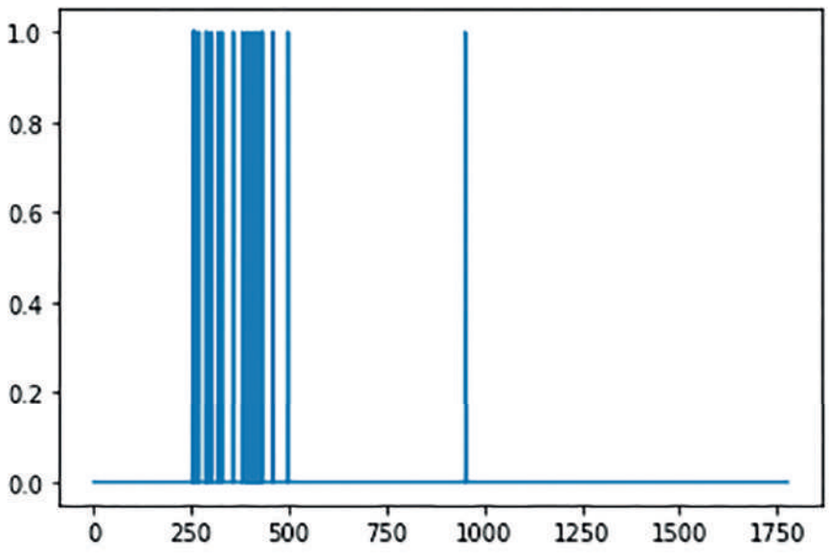


If you plot any of the X, Y, or Z fields, or all of the data, you will hopefully see some patterns. All of ours showed a significant jump of approximately 300 timepoints around when the flush was made. There are some anomalies, which is only to be expected with real sensors—we had a jump at around about the 950 timepoint, which could have been anything from a door shutting to a pipe vibrating. What we need to do is extract the "flush" data as a "detection" and ignore the rest. This is a common practice, as many signals suffer from noise, whether it's a mobile-phone tower interfering with satellite signals, or in this case everyday movements of the air, passing traffic, etc. interfering with our accelerometer readings.

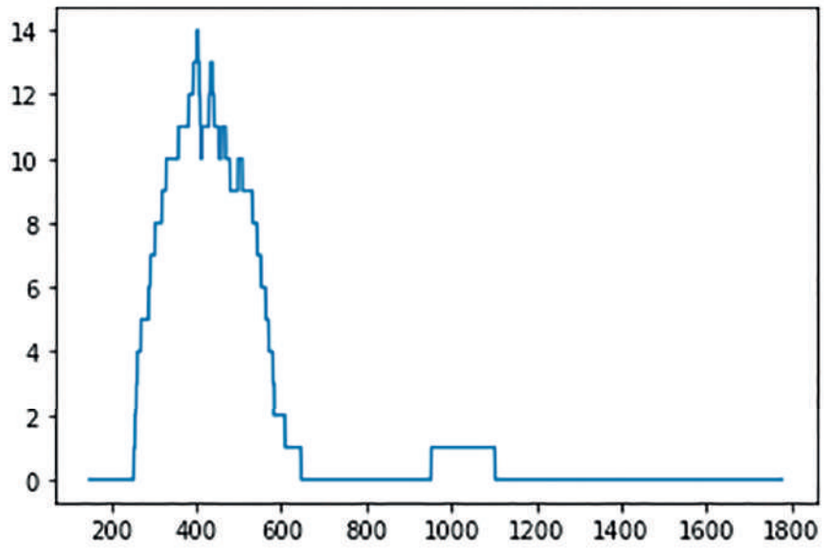
To do this we can add an "Average" series to the data set, which works out a mean of the X, Y, and Z numeric columns. Plotting this shows a slightly more usable plot. We can then use NumPy to add a Boolean flag, **AboveThreshold**, which will put a 1 for whenever we **think** an acceleration (or vibration) is from a flush due to its severity and a 0 when it is not. By looking at our plots, we chose 0.005 as the threshold. The code looks like this: `data['AboveThreshold'] = np.where(data['Average'] > THRESHOLD,1,0)`.

Plotting the thresholds gives us a more usable chart, although we are still seeing some jumping, and one anomaly. Adding a rolling sum of the above threshold points gives us a bit more of an idea of what we're working with.

```
Out[26]: <matplotlib.axes._subplots.AxesSubplot at 0x7ff77b2d6700>
```



```
Out[27]: <matplotlib.axes._subplots.AxesSubplot at 0x7ff77b398f10>
```



This rolling sum has, crucially, negated the problem of the anomaly as, since it didn't last long, it becomes less important. It is then reasonably simple to step through the data using a **for** loop, detect any counts that we think trigger a flush (we said `>= 13` in the rolling sum), check that we don't think we're currently already in a flush period, and highlight this to the user.

Testing

If you start recording and then place the sensor, you may find just putting it down and attaching it gives quite a lot of false data which can then prove hard to remove. We attached the sensor to the water inlet pipe with some electrical tape and **then** plugged the battery in to start things off. It is also worth keeping the data quite "staged" to start with so that you know what to look for and are able to investigate anomalies before trialing with real data.

Stretch tasks

- Many lavatories have short and long flushes. Modify your model to be able to differentiate between the two. This is a good opportunity to have conversations with your family about wasting water.
- The data, as you have probably noticed, is quite large. Modify the program to reduce the amount of data stored by either storing it as a binary file using two's complement (so only two bytes would be needed per reading), or by multiplying/truncating the data readings so that fewer digits need to be stored.
- The website https://worldtimeapi.org/ provides a simple, plain text way of checking the current time. Modify your program to store actual time codes rather than timepoints.

- Your Raspberry Pi Pico has two cores. At the moment when you request data, the "main" core is being blocked from taking more readings due to having to send the data. Modify your code to use threads to send the data via the second core.

Final thoughts

There are many different ways to use sensors that you may not think about. Using an accelerometer for measuring vibrations of pipes is just one of them. This project really highlights the wastage of resources that is prevalent within our societies. It would be a good opportunity to spend some time discussing your findings from this project with your family, and then considering what other things you could monitor in this way. Tap usage could be monitored (either to make sure people are washing their hands, or to detect drips and leaks) to name just one other way we can work toward Global Goal 6 and manage water sustainably.

7. EXPERIMENTING WITH PHYSICS

Setting the scene

This book has spent a significant amount of time discussing monitoring waste; another way of taking more care of our planet is to do things better in the first place. Part of this is measuring and recording the efficiency of things, whether it's the air resistance of the latest fighter jet or the efficiency of a new battery-powered car engine. Innovative methods can feed into a better, more productive, and ultimately more ecologically-friendly industry, supporting Global Goal 9.

Microcontrollers are particularly good for monitoring and measuring systems as they are incredibly small, light, and low powered and therefore have a minimal impact upon devices they are attached to. In this project we will be attempting to monitor and record the drop of a parachute using a time-of-flight sensor. We will take you through the Computer Science aspects of the system—the physics is a bit trickier so you may need to spend a bit more time applying the data to the models. Time-of-flight sensors can be used to measure air resistances in projects such as this, physical growth rates, accurate distances between objects, and a whole host of other uses which can aid industry in streamlining and improving manufacturing technology.

Success criteria

- ☑ Connect a time-of-flight sensor to the Pi Pico W and test it.
- ☑ Connect and detect button presses from a GPIO pin.
- ☑ Develop a finite state machine and move between states.
- ☑ Record data into individual model files.
- ☑ Set up the code for running offline and build a physical device.

1

This project makes use of the VL53L1X—a time-of-flight sensor. The flight being measured isn't the device flying through the air, but rather the time it takes a low-powered laser to fly to an object and back again, allowing a very effective and accurate distance measurement. There is a powerful and simple-to-use library for the VL53L1X. Unfortunately, it is not currently possible to install it on the Pico. Thankfully, a British engineer, Lee Halls, has cloned and modified part of the library, which we have modified slightly and made available as **vl53l1x.py** on the GitHub repository. You will need to copy this to your Pico.

```
from machine import I2C #Gain access to the I2C port
from vl53l1x import VL53L1X #Import the ToF library
import time
#Set up the i2c connection
sda=machine.Pin(20)
scl=machine.Pin(21)
i2c=machine.I2C(0,sda=sda, scl=scl, freq=100000)

distance = VL53L1X(i2c) #Start the ToF sensor
while True:
    print("range: mm ", distance.read()) #Output the distance reading
    time.sleep(0.05) #Pause
```

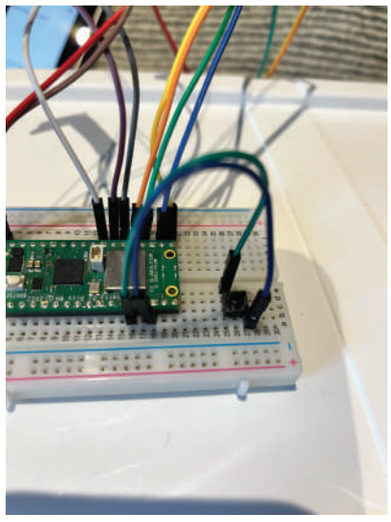
The sample code in `ToF_Test` will ensure the sensor is working. You can plug it into the breakout in the Pico Explorer for testing for now if you wish, although we soldered headers to it (coming out of the back) for connection to the expander we will be using later. Once the sensor is running, `.read()` can be used very simply to measure the distance in millimeters.

PRO TIP

The VL53L1X comes with a little piece of film over the sensor to protect it during transit. If your measurements are not coming out right, make sure you have removed it.

2

We are going to make use of a button to start recording. As we are continuing to use the SD card, you will attach a button to the breadboard instead of using the buttons on the explorer board. We have attached the button to Pin 12 (for reference this is button A on the Pico Explorer board). The button is configured as an input device and set with a pull up resistor. This means, in its resting state, the button value will read as a value of 1 or as True. A quick Internet search will provide images of the Pico pin out diagram. One side of the button is connected to Pin 12 and the other to any of the ground pins on the Pico.



Most buttons have no polarity, so it doesn't matter which leg is connected to ground and which is connected to the signal pin. In the main loop of the code, we check the `button.value()` and if it reads as 0, the button has been pressed. The program then transitions from the **READY** state to the **COUNTING** state.

To handle potential errors gracefully, the entire main loop is wrapped in a **try... except construct**. If an error occurs (if the sensor is detached or the SD card fails) the code in the **except** block is executed. This approach ensures the program doesn't crash, maintaining stability even in unexpected conditions—a critical feature for microcontroller-based projects.

PRO TIP

Motherboard manufacturers have become masters in communicating with binary outputs, either LEDs or buzzers. A combination of fast and slow flashes or differing colors can tell a user a lot. It is worth not only forming a series of instructions based on codes or colors, but writing them down and communicating them.

3

We looked at finite state machines in an earlier chapter. There are again three distinct states:

- **READY** (State 0): The device waits for the button press to start recording. The LED turns on.
- **COUNTING** (State 1): The device measures distances at regular intervals. When the distance falls below a predefined threshold (indicating the object has landed), it transitions to the **SAVING** state. The LED flashes.
- **SAVING** (State 2): The data collected during the **COUNTING** state is saved to the SD card. Once saving is complete, the device returns to the **READY** state. The LED once again turns on while saving data.

```
# Button Setup
button = Pin(12, Pin.IN, Pin.PULL_UP)

# State Variables
READY = 0
COUNTING = 1
SAVING = 2
state = READY
led_status = False
start_time = time.ticks_ms()
distances = []

# Main Loop
print("Logging distance data and serving over HTTP...")
while True:
    try:
        dist = distance.read()

        if state == READY:  # Wait for button press to start recording
            if button.value() == 0:
                print("Starting measurement...")
                distances = []
                state = COUNTING

        elif state == COUNTING:  # Record distances until close enough
            elapsed_time = time.ticks_diff(time.ticks_ms(), start_time)
            distances.append((elapsed_time, dist))
            print(f"{elapsed_time}ms -> Distance: {dist}mm")

            if dist < 10:  # Save if object is detected within 10mm
                state = SAVING

        elif state == SAVING:  # Save data to SD card
            print("Saving data to SD card...")
            with open(FN, "a") as f:
                for item in distances:
                    f.write(f"{item[0]},{item[1]}\n")
            state = READY

        # Handle HTTP requests
        client, addr = server.accept()
        print("Client connected from", addr)
        request = client.recv(1024).decode("utf-8")
        if "/data" in request:
            with open(FN, "r") as f:
                response = "HTTP/1.1 200 OK\r\nContent-Type: text/plain\r\n\r\n"
                response += f.read()
            client.send(response)
        else:
            client.send("HTTP/1.1 404 Not Found\r\n\r\n")
        client.close()

        time.sleep(0.1)

    except Exception as e:
        print(f"Error: {e}")
        break
```

As there is a chance the sensor does not hit the ground, the change from **COUNTING** to **SAVING** happens when the distance is at 10mm from the ground. This can be altered depending on the sensitivity required. The periodic distance measurements during the **COUNTING** state are handled independently from the delays between the state transitions, ensuring precise data capture.

4

Unlike previous implementations where the data file was overwritten with each restart, this version allows multiple experiments before transferring data. Files are named sequentially (e.g., 0.csv, 1.csv, etc.), and the program uses `uos.stat()` to check if a file exists. If a file exists, the counter is incremented until an unused filename is found. This ensures that data from each experiment is saved independently and is easy to identify later.

The SD card functionality, combined with exception handling and state-based operation, makes this system robust, adaptable, and easy to use for real-world experiments.

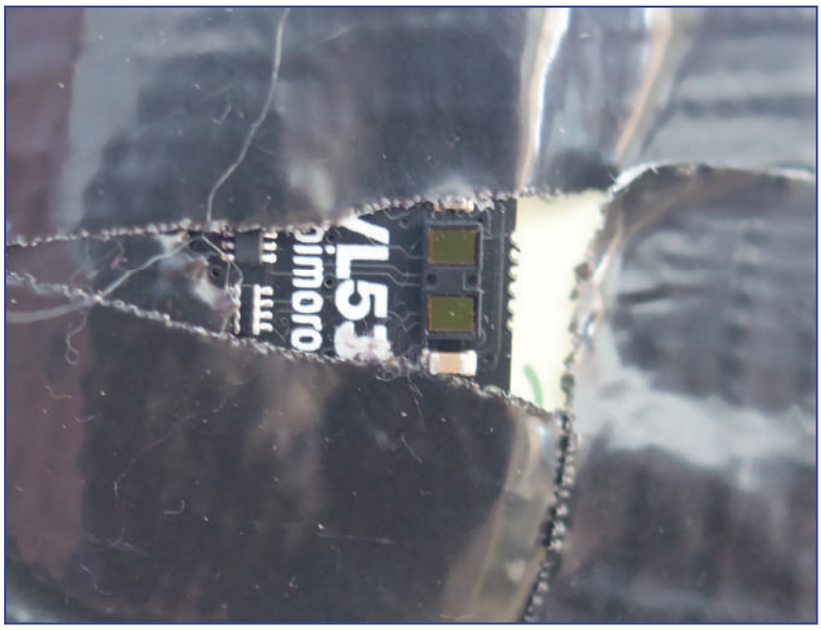
```
from machine import I2C, Pin, SPI
from vl53l1x import VL53L1X
from sdcard import SDCard
import uos
import time
import network
import socket

# Wi-Fi Configuration
SSID =
PASSWORD =

# I2C Setup for VL53L1X
sda = Pin(20)
scl = Pin(21)
i2c = I2C(0, scl=scl, sda=sda, freq=100000)

# SPI Setup for SD Card
sck = Pin(18)
mosi = Pin(19)
miso = Pin(16)
cs = Pin(5)
spi = SPI(0, sck=sck, mosi=mosi, miso=miso)

# Initialize SD Card
try:
    print("Initializing SD card...")
    sd = SDCard(spi, cs)
    uos.mount(sd, "/sd")
    print("SD card mounted successfully!")

    # Create a new file and write the header
    FN = "/sd/distance_data.csv"
    with open(FN, "w") as f:
        f.write("ms,distance\n")
except OSError as e:
    print(f"SD card error: {e}")
    raise SystemExit()

# Wi-Fi Setup
def connect_to_wifi():
    wlan = network.WLAN(network.STA_IF)
    wlan.active(True)
    wlan.connect(SSID, PASSWORD)
    print("Connecting to Wi-Fi...")
    while not wlan.isconnected():
        time.sleep(1)
    print("Connected to Wi-Fi!")
    print("IP Address:", wlan.ifconfig()[0])

```

```
connect_to_wifi()

# HTTP Server Setup
def start_server():
    addr = socket.getaddrinfo('0.0.0.0', 80)[0][-1]
    s = socket.socket()
    s.bind(addr)
    s.listen(1)
    print("Listening on", addr)
    return s

server = start_server()

# Initialize VL53L1X Sensor
distance = VL53L1X(i2c)

# Button Setup
button = Pin(12, Pin.IN, Pin.PULL_UP)

# State Variables
READY = 0
COUNTING = 1
SAVING = 2
state = READY
led_status = False
start_time = time.ticks_ms()
distances = []

# Main Loop
print("Logging distance data and serving over HTTP...")
while True:
    try:
        dist = distance.read()

        if state == READY:  # Wait for button press to start recording
            if button.value() == 0:
                print("Starting measurement...")
                distances = []
                state = COUNTING

        elif state == COUNTING:  # Record distances until close enough
            elapsed_time = time.ticks_diff(time.ticks_ms(), start_time)
            distances.append((elapsed_time, dist))
            print(f"{elapsed_time}ms -> Distance: {dist}mm")

            if dist < 10:  # Save if object is detected within 10mm
                state = SAVING

        elif state == SAVING:  # Save data to SD card
            print("Saving data to SD card...")
            with open(FN, "a") as f:
                for item in distances:
                    f.write(f"{item[0]},{item[1]}\n")
            state = READY

        # Handle HTTP requests
        client, addr = server.accept()
        print("Client connected from", addr)
        request = client.recv(1024).decode("utf-8")
        if "/data" in request:
            with open(FN, "r") as f:
                response = "HTTP/1.1 200 OK\r\nContent-Type: text/plain\r\n\r\n"
                response += f.read()
            client.send(response)
        else:
            client.send("HTTP/1.1 404 Not Found\r\n\r\n")
        client.close()

        time.sleep(0.1)

    except Exception as e:
        print(f"Error: {e}")
        break
```

5

We have already done a lot of the work in preparing to run detethered. We know we can rename the file **main.py** on the Pico so that it will run automatically. We have wrapped the whole loop inside a **try…except...** loop and included error codes via turning on and off and flashing the on-board LED at various stages of the code execution so we know what is happening. We have triggered the change in state from the button and stored to the MicroSD card.

However, setting up the device is a bit more dependent on what methodology you wish to use. We chose to recycle an old olive spread tub by (not very) carefully cutting a hole in the bottom and poking the sensor (on wires) through the hole. As we soldered the headers on backward, the sensor sat nicely against the bottom of the pot. The expander holding the Pico W and the wires for the sensor were carefully stuck to one side of the pot with strips of gaffer tape, and the battery was stuck in the same way to the other. More strips were used to hold the sensor in place and stop the wires falling out when the device hit the ground. While not very pretty, the device works fine for prototyping and a large piece of artboard was attached with string to form a flat "parachute" for us to experiment with.

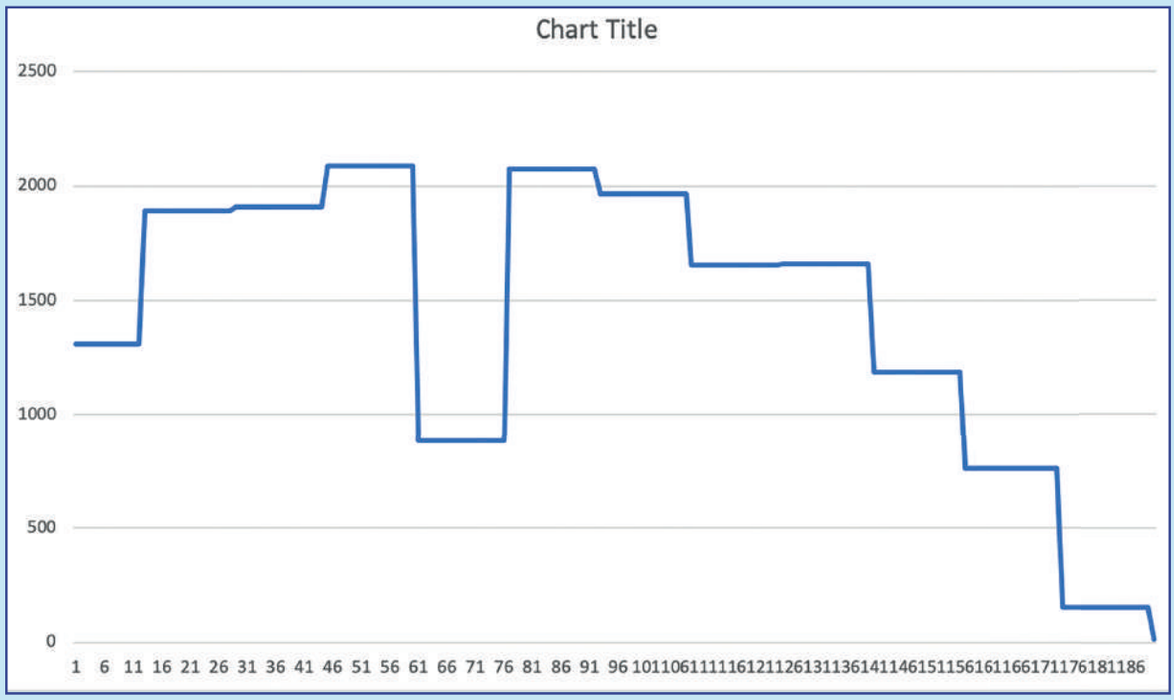


PRO *TIP*

Running detethered (where the Pico runs automatically and is not connected to a controlling computer) can sometimes lead to it no longer responding (or not being detected when you do plug it in). If this is the case, and re-starting does not help, there is a flash_nuke.uf2 file (available online from AdaFruit if you search for the file name) which you can drop onto the Pico. You will need to re-install Circuit Python and the Python files afterwards.

Testing

You will probably want to run quite a few tests before you're satisfied, adjusting the recording period and the way you carry out the experiment. You can see an odd drop on our graph below (which we created by opening the data in Excel and creating a line graph); this was due to a rogue elbow getting in the way. We also experienced the USB cable flopping around, the sensor "basket" landing on its side, and nosey dogs getting in the way. This is normal when testing prototype devices, so don't despair. Once you have reliable readings and a positive method, try varying things such as the parachute material, size, extra weights, and so on to develop different models.



Stretch tasks

- Connect your Pico W to WiFi and automatically serve the latest CSV file to a notebook in Jupyter on demand.
- You should be able to calculate overall speed for each "drop"; have the Pico calculate this and append this and any other statistics you feel are useful to the bottom of the CSV file.
- Have the Pico automatically cleanse the data by removing obvious errors.
- When you have collected enough data for a particular parachute, analyze the data sets in Jupyter Notebook and, using the Internet to assist you, try and calculate the drag coefficient for the parachute in question.

Final thoughts

There are numerous uses for time-of-flight sensors. By gaining more insight using sensors such as these, we are able to alter the way we develop and manufacture all sorts of things around the world. LiDAR (Light Detection and Ranging), using time-of-flight sensors, is used in automated vehicles, for example, to provide automated braking/acceleration based on traffic states. Not only does this improve safety but it can also be used to improve the fuel efficiency of vehicles and ultimately has a positive benefit on the planet. In automation industries, time-of-flight sensors can be used to measure distances accurately. One simple example is to measure the "rise" of bread dough, ensuring that bread can be baked as soon as it has risen enough. This reduces waste in baking bread that has not fully risen and has to be destroyed, as well as reducing the need to heat proofing ovens in which the bread is already fully risen. The innovative use of a low-price sensor can save costs across a whole industry and is just one example of an industry working toward Global Goal 9.

8. SECURITY STARTS AT HOME

Setting the scene

Security of data is an increasingly important consideration. In 2020 there were over 1 billion personal records compromised in the U.S. alone through username/password attacks—an increase of 450% over the year before. In the U.K. in the first half of 2020 the staggering amount of nearly 4 billion records were breached through 815 data attacks. There are numerous ways of protecting data, from secure passwords to encryption and air gaps, but the most often overlooked type of attack is the physical attack on a system, and within this the most prevalent is an attack caused by an unaccounted visitor. Innovative ways of solving problems such as this are what drives new industry and expanding infrastructure and links with Global Goal 9.

In this project you will explore some new hardware, the HuskyLens, to use Artificial Intelligence (AI) to carry out facial recognition of new visitors (this could be to your front door, or even your bedroom, depending on where you keep your most important data). You will also be introduced to the *IFTTT*—if this then that—concept, which will enable notifications on your mobile phone.

Success criteria

- Connect and test a HuskyLens to the Pi Pico W.
- Train the HuskyLens to identify multiple faces and push the identification to the software.
- Identify faces of concern in the application and log to the SD card.
- Raise an IFTTT notification on a mobile phone.

1

This project makes use of a HuskyLens—a remarkable piece of equipment which can, out of the box, conduct a number of image recognitions including faces, objects, colors, and line following. When you open your HuskyLens the first thing you should do is update the firmware as the chances are it is not on the latest version. The HuskyLens wiki document has links to everything you need—in Windows you:

- download the latest firmware;
- download and install a USB to UART driver so your computer can communicate with the lens; and
- download the K-Flash software and use it to install the firmware on the HuskyLens.

The process takes 5–10 minutes and only has to be done once. It is much easier on a Windows computer than a Mac or Linux computer, so as a one-off it may be worth borrowing a friend's computer for the process or exploring the use of a Windows Virtual Machine. It is possible with a Mac as well, so if you don't have access to a Windows computer you will get through with perseverance. Before using the lens with the pico you will need to copy the required libaries to the Pico. These are included in the GitHub folder. For a step-by-step guide of how to install these follow this simple guide by Husky Lens: **https://community.dfrobot.com/makelog-311712.html**

Once we have the HuskyLens updated we have two things to do: we need to train it to recognize a face, and we need to have it tell Python that it has done so. To connect the HuskyLens to the Pico we are again using the Pico Explorer. The HuskyLens comes with a 4-pin wire, so plug it in and connect the wires to the Pico W via the Pico Explorer board.

- red goes to the + or 3 v pin
- black goes to the – or ground pin
- green goes to GP2 on the Pico Explorer
- blue goes to GP3 on the Pico Explorer

To get everything freshened up (in case, like us, you just couldn't resist playing with it!) press the **selection** (rotating) button and scroll across to **General Settings**. Press it again and scroll across to **Factory Reset**, then press twice.

PRO *TIP*

If things get a bit confused, the factory reset can always be used to start afresh, so do feel free to explore the HuskyLens. The firmware will not reset so you don't need to re-install that.

You should now make sure that it is in Face Recognition mode. Press the **selection** button, scroll to **Face Recognition** if needed, and press again. Now when you hold the camera up to a face, you should see a gray box and the word "Face". We found it convenient to print off a picture of Stephen Fry for testing purposes. When you see Stephen's face in the gray box, there should be a yellow crosshair in the middle. Press the **Learn** button once to learn Stephen's face. There will be a brief pause and then the box will turn blue and indicate that Stephen's face has been learned.

2

Now that we know the HuskyLens is working it is time to get it set up for our project. Forget the image you have learned (or perform a factory reset). You need to enter "multiple" mode, so long press the **selection** button—it will flick through **Face Recognition** and go to **Learn Multiple**. Press again to select, and rotate so the slider turns blue and moves to the right. Press once more, rotate to save and return, and press to exit. Now you are in multiple mode, point toward the first face you want to learn (we used Stephen Fry again) and long press the **Learn** button. When learned, press again and long press over another face (we used Hugh Laurie's photo this time). If you are creating a security system for your bedroom, learning your family's faces might be a good idea!

When you hover over a known face, a colored box and ID number should show up. When you hover over an unknown face (we used a picture of Rowan Atkinson as a control), a gray box, and no ID number will show.

We have included a library **huskylensPythonLibrary.py** in our GitHub repository which you can copy to your Pico to be able to use the HuskyLens. You will have to import **HuskyLensLibrary** from it and create an object (we called it **husky**) with a parameter of **I2C**.

PRO *TIP*

If the HuskyLens won't connect, try changing the protocol setting to I2C instead of AutoDetect. Don't forget to save!

3

The HuskyLens has a handy **knock** command to ensure it's working.

```
from huskylensPythonLibrary import HuskyLensLibrary

husky = HuskyLensLibrary("I2C")  # Use the same I2C setup
print(husky.command_request_knock())  # Should return "Knock knock
```

Once this is working, we can see about getting some useful information from the HuskyLens. Calling `command_request()` will return a list. If the list is empty, the HuskyLens has not detected any faces. If it is not empty, it will contain one or more lists, each of five elements. Each list is a face that has been detected. The first four elements are positional elements of the face, and the fifth is the ID of the face. If the ID is 0, then the face is unrecognized and if it is non-zero, then it is the ID of an identified face. For now, we can pass this ID to a `faceDetected()` function and just output the name of the face owner.

```
from huskylensPythonLibrary import HuskyLensLibrary
import time

husky = HuskyLensLibrary("I2C") #Create the HuskyLens object as I2C


#Deal with detected faces
def faceDetected(faceId):
    if faceId == 0: #Face 0 is unknown
        print("Unknown!")
    elif faceId == 1: #Known face
        print("Stephen Fry")
    elif faceId == 2: #Known face
        print("Hugh Laurie")
    else: #A known face that we've not yet programmed!
        print("You've been learning new faces!")

print(husky.command_request_knock()) #Knock knock - is it working?
while True: #Work forever
  result = husky.command_request() #Get the details of any faces
  #print(result)
  numFaces = len(result) #Check how many are identified
  if numFaces > 0: #If the list isn't empty, there are faces.
      for face in result: #Go through each face
          faceId = face[4] #Get the face ID
          faceDetected(faceId) #And deal with it
  time.sleep(1) #Run once a second

```

4

At the moment, we don't do anything different with known or unknown faces. We can easily adjust the function `faceDetected()` to log any unknowns to the SD card on the HuskyLens by calling `command_request_screenshot()`. This will produce an error ("Read error") but that's an implementation problem with the library and we will have to ignore it and assume everything works.

There is a bit of a problem, however: quite often a known face will be mis-identified as it moves in/out of frame. Instead, we will create two timers, with 5-second boundaries. If we see an unknown face for more than 5 seconds without seeing a known face, we can take a photo to explore later.

PRO *TIP*

The MicroSD slot on the HuskyLens is a little confusing. With the HuskyLens screen facing down and the MicroSD slot toward you, place the card with the writing facing upwards and the gold contacts down and toward you, then slide the card toward you into the socket.

```
# Timer thresholds in seconds
UNKNOWN_TIMESPACE = 5  # Time to wait before confirming an unknown face
KNOWN_TIMESPACE = 5    # Time to wait before confirming a known face

# Initialize timers
timeUnknown = None  # Timestamp for when an unknown face is first detected
timeKnown = None    # Timestamp for when a known face is last detected

# Function to handle detected faces
def handle_detected_face(faceId):
    """
    Handles actions based on the face ID detected by HuskyLens.
    :param faceId: ID of the detected face
    """
    global timeUnknown, timeKnown

    if faceId == 0:  # Face ID 0 indicates an unknown face
        print("Unknown face detected!")
        if timeUnknown is None:
            timeUnknown = time.time()  # Record the first time an unknown face is detected
        if timeUnknown and (time.time() - timeUnknown) >= UNKNOWN_TIMESPACE:
            print("Unknown face confirmed after 5 seconds!")
            timeUnknown = None  # Reset the timer
    elif faceId in [1, 2]:  # Face IDs 1 and 2 correspond to known faces
        if faceId == 1:
            print("Hello, Stephen Fry!")
        elif faceId == 2:
            print("Hello, Hugh Laurie!")
        timeKnown = time.time()  # Update the known face timer
        timeUnknown = None      # Reset the unknown timer
    else:  # New or unprogrammed face IDs
        print("New face detected! HuskyLens is learning new faces.")
        timeKnown = time.time()  # Treat new face as a known face temporarily

```

5

We're unlikely to want to sit and watch our computer screen all of the time just in case Rowan Atkinson is trying to steal all our data from our bedroom. We have looked at pulling data from the Pico, and pushing to an endpoint for data analysis, but we haven't yet explored another excellent IoT concept: IFTTT. IFTTT, which stands for "if this, then that", allows you to create webhooks and applets that react to those webhooks. Simply, we call a URL (the webhook), the IFTTT applet sees that URL has been called, and does "something". In this case, it can be a pop-up notification on your mobile phone.

You will need to sign up for a free account at **ifttt.com** then go to **ifttt.com/maker_webhooks** and create a webhook. Keep things simple with a web request and give it a name (we went for "Unknown_Visitor"). For the "then that", create a notification. Don't forget to add the IFTTT app to your mobile phone and sign in with the same account. Once set up, you can click on your icon and then **My Services**, scroll down to **WebHooks** and, oddly, click on **Documentation** to get the URL you need to post to. You need to replace **{event}** with your request name ("Unknown_Visitor" for us) and copy the URL into your Python project.

In Python, you can copy the same **HTTP_** constants as you used in the **4.1wifitest.py** project (used in *4. The "I" in IoT*) and reuse the handler, although IFTTT doesn't seem to return 200 so we checked for "Congratulations" in the body to confirm it worked. You can start the WiFi connection when you start the program, then just call the `http_request()` whenever you decide to save an image.

```
import network  # For WiFi connection
import urequests  # For HTTP requests
import time  # For delays
from machine import Pin # For controlling the onboard LED

# Wi-Fi credentials
SSID = "Your ssid"  # Replace with your WiFi SSID
PASSWORD = "your password"  # Replace with your WiFi  Password

# HTTP Settings
HTTP_REQUEST_HOST = "http://google.com"

# Onboard LED setup
led = Pin("LED",Pin.OUT)  # Built-in LED on Pico W

# Connect to WiFi
def connect_to_wifi():
    wifi = network.WLAN(network.STA_IF)
    wifi.active(True)
    led.off()  # LED off while connecting
    print("Connecting to WiFi...")

    wifi.connect(SSID, PASSWORD)

    while not wifi.isconnected():
        time.sleep(0.5)  # Wait until connected
        print("Connecting to WiFi...")

    print("Connected to WiFi!")
    print("IP Address:", wifi.ifconfig()[0])
    led.on()  # LED on when connected

# Perform an HTTP GET request
def make_http_request():
    try:
        response = urequests.get(HTTP_REQUEST_HOST)
        print("HTTP Response Code:", response.status_code)
        print("Response Content:", response.text[:500])  # Show first 100 characters
        response.close()
    except Exception as e:
        print("HTTP Request Failed:", e)

# Main Program
connect_to_wifi()
make_http_request()

```

```
38
39          if time.time() - timeUnknown >= UNKNOWN_TIMESPACE:
40              print("Unknown face seen for too long. Sending alert...")
41              print(husky.command_request_screenshot())  # Attempt to capture a screenshot
42
43              # Turn on onboard LED for 5 seconds
44              led.on()
45              LEDOnTime = time.time()  # Store when LED was turned on
```

We also included a timer so we can turn off the sent/not-sent LED (light-emitting diode) after 5 seconds.

Testing

AI is not an exact science. Testing with printed-out images is very helpful, but doesn't deal with different angles, lighting conditions, or funny faces. Recruit some members of your family and make sure you test all of these considerations fully.

Stretch tasks

- The "dealing with problems" algorithm—not saving a photo if a known face is seen within five seconds—is quite rough and ready. Develop your own algorithm and test it with family members. You should also decide what to do should there be multiple faces in view at once.
- The IFTTT protocol allows you to include a payload in the notification. Include a timestamp and more descriptive message.
- Explore the HuskyLens library file: there is a command to instruct the lens to learn a new face. Program the HuskyLens to learn a new face when a button on the Pico Explorer is pressed.
- Log all data—known and unknown faces, photos saved, notifications sent, etc. to the SD card connected via SDI (unfortunately the one on the HuskyLens can't be accessed or read without putting into a computer). Put the data into Jupyter Notebook to analyze and identify key times that unknown visitors were detected.

Final thoughts

The tremendous power available on microcontrollers (which is what the Pico and the HuskyLens are) is changing the world of AI and Machine Learning. These developments are only going to expand over the coming years. While there are some fears about AI run rampant, its power for transforming manufacturing, detecting cyberattacks, identifying trends, and many other modern-day problems is just starting to become apparent. Devices such as the HuskyLens let people like you get involved at this exciting turning point. Exploring innovative opportunities to use Machine Learning to enhance industry and infrastructure is a Global Goal, and with good reason as opportunities arise to make the world a far better place. But with this potential for good comes potential for bad. Spend some time identifying and thinking about some of the ethical issues that may come about, and indeed have already come about, from deploying facial recognition software.

9. BRINGING IT ALL TOGETHER

Setting the scene

Microcontrollers are being used in all sorts of situations, from monitoring traffic to controlling manufacturing, and a multitude of areas in between. Being able to see a problem and plan an end-to-end solution is a key skill to develop and is invaluable when developing innovative new solutions for industry and infrastructure, Global Goal 9.

In this project we are going to take on the challenge of squirrels vs. birds. This is a common problem. Many birds benefit from having their diets supplemented over the winter by people putting out bird food, as a lot of their natural habitat (and food supplies) have been decimated by development. Squirrels, which are often thought of as pests, have got wise to this and have a tendency to visit the bird feeder and steal the food before the birds can get to it.

In the previous chapters we have walked you through some problems and provided working solutions. In this final chapter we will discuss potential solutions to the problems faced here, but will not be providing worked code or solutions; this one is up to you. You can of course refer back to the previous chapters, and how problems have been solved there, to guide you and suggest some ideas for the solutions!

Success criteria

- ☑ Decompose the problem to identify sub-problems: how to feed the birds, how to identify when food needs re-filling, and how to scare away squirrels.
- ☑ Identify hardware to solve the identified sub-problems.
- ☑ Plan and build a complete solution.

1

The overall problem we have is that we want to feed the birds, but the squirrels keep stealing the food. This is a relatively complex problem, so it makes sense to decompose the problem to sub-problems. We could consider these two sub-problems:

1. How to feed the birds.
2. How to **not** feed the squirrels.

These problems could be further broken down:

1. How to feed the birds:
   a. identify that a bird needs feeding;
   b. dispense the food; and
   c. know when the food needs re-filling.
2. How to not feed the squirrels:
   a. detect that a squirrel has appeared; and
   b. scare the squirrel off.

Each of these sub-problems can then be tackled individually. In a larger system you would allocate each sub-problem to an individual or team.

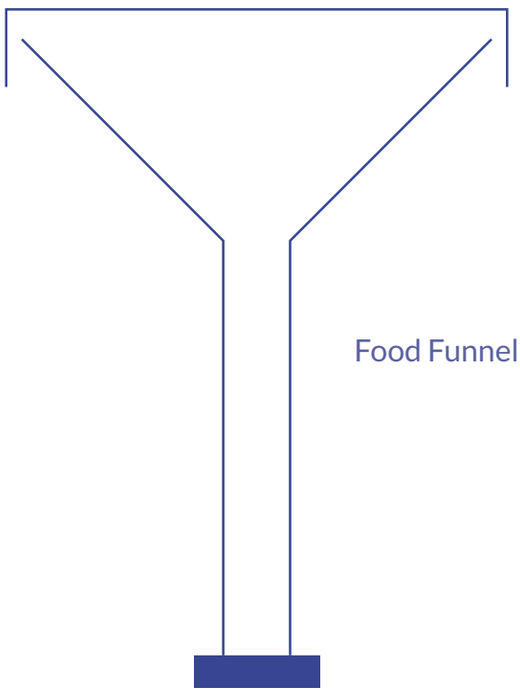
PRO *TIP*

Decomposition is an essential skill not just in Computer Science, but in all areas of life. Whenever a problem seems too big, breaking it down can make it easier to solve. In Computing we break things down into modules and subroutines.

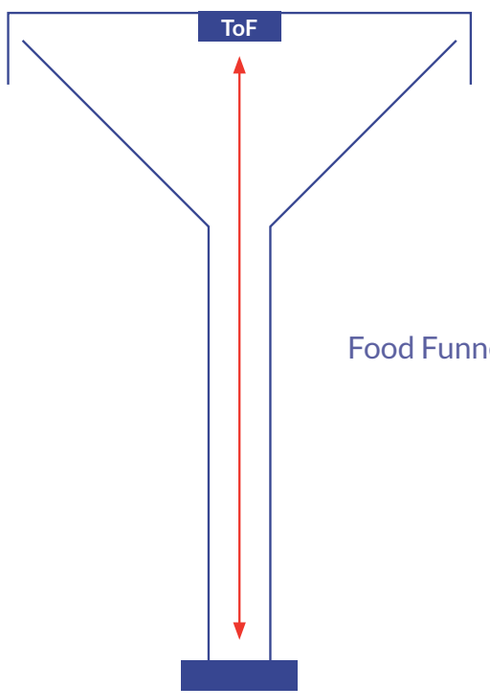
2

Two of our problems could be solved together. The HuskyLens has another mode; it can be used for object detection. By training it to recognize "birds" and "squirrels" it can be used to trigger differing parts of the code.

Some of the other problems are a bit trickier. Let's consider delivering food. Rather than re-inventing the wheel, we can seek inspiration from other automatic feeders. Fish feeders such as the "Fish Mate" have a rotating disk with little pots of food that rotate over a hole and release food into the water. We could do something similar, with a tube or funnel with food in and a cap which rotates out of the way to drop some food onto the table when a bird is detected.



Food Funnel



Food Funnel ToF

We could extend this solution to also solve the problem of knowing when the food needs to be re-filled. When the food is empty, there will be a gap between the lid of the food storage and the removable cap; we can measure that distance using a device such as a time-of-flight sensor. This could report home via WiFi or using visual clues such as a flashing LED (light-emitting diode).

Scaring squirrels off is not an exact science, but it is a fairly safe bet that a loud noise would do the job. We have already explored the use of a piezo buzzer on the Pico Explorer base unit. Alternatively, there is an audio module that you can plug your Pico into and connect to speakers.

Of course, a waterproof housing and some safe way of powering the device would be very useful. We will leave this to you to work out but do remember: electricity and water do not mix.

3

You have already met most of the hardware you will need for this project, should you choose to implement it in the same way as us. The Raspberry Pi Pico W is a given; you have explored using the WiFi capability for communication and storing data on an SD card in previous chapters. The HuskyLens can be used in object detection mode to detect birds and squirrels. If you want to use a passive piezo but not the Pico Explorer base, these are inexpensive and just need connecting to ground and a GPIO pin. They can be controlled by pulse width modulation, with a *frequency* being set (the note) and then a *duty cycle* (a ratio of how long it is on vs how long it is off). Alternatively, there are a number of speaker options available as add-ons for the Pico W which will allow for a wider range of sounds to be output.

```
from machine import Pin, PWM
from utime import sleep_ms

buzzer = PWM(Pin(15))
buzzer.freq(500)
buzzer.duty_u16(1000)
sleep(1)
buzzer.duty_u16(0)
```

The one piece of hardware you have not yet met is the servo. A *servo motor*, or just *servo*, is a rotary actuator. It can rotate an arm one way or another to, in our case, block or unblock the food dispenser outlet. Controlling the servo is similar to the way the buzzer works. It needs connecting to a GPIO pin (and + and −) and is controlled by setting an initial frequency (50) and then modifying the duty (in this case in nanoseconds). By setting the duty cycle to differing (pre-defined) constants, we can adjust the rotational position of the servo.

The following code will move the servo between its minimum, maximum, and middle positions.

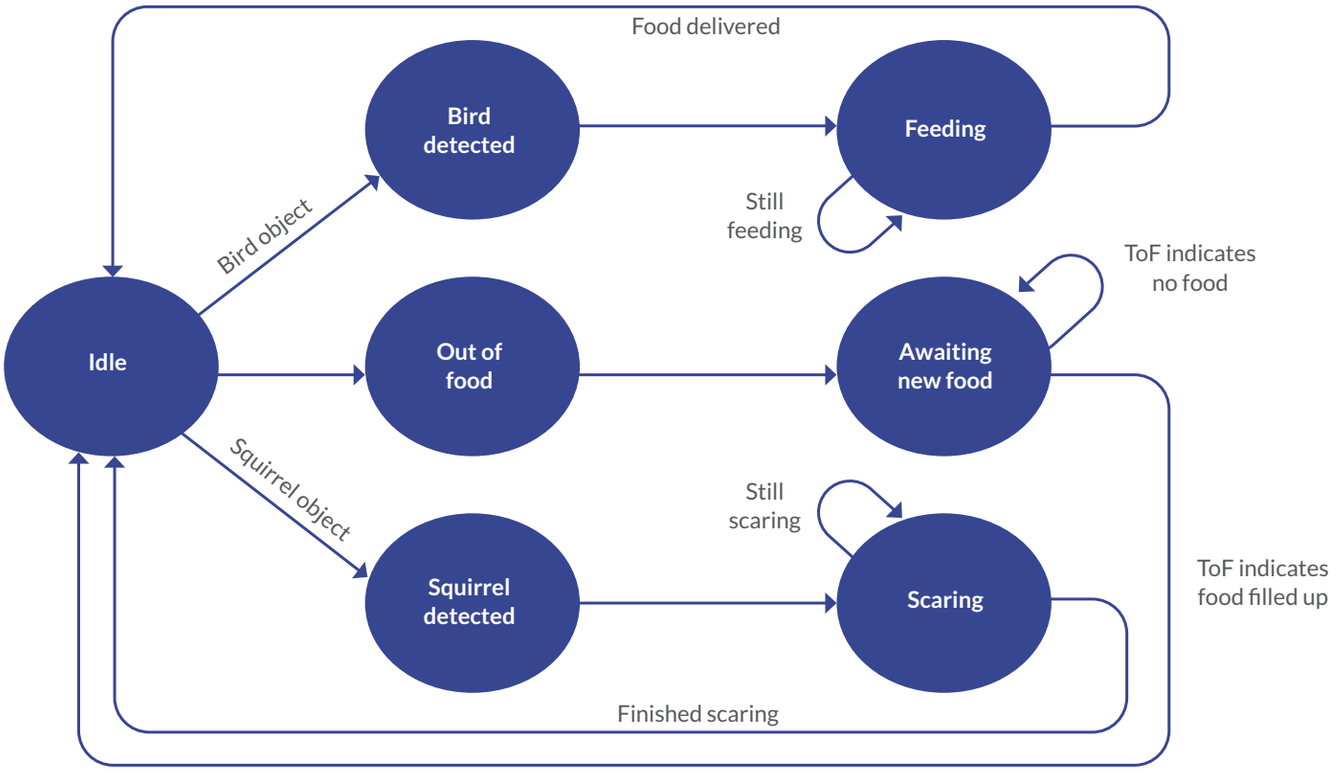
```
from machine import Pin, PWM
from utime import sleep_ms

MID = 1500000
MIN = 1000000
MAX = 2000000
SERVOPIN = 15
SERVOFREQ = 50

servo = PWM(Pin(SERVOPIN))
servo.freq(SERVOFREQ)
servo.duty_ns(MID) #Initialize to middle position
sleep(1)

while True:
    servo.duty_u16(MIN) #Move to minimum
    sleep(1)
    servo.duty_u16(MID) #Back to middle
    sleep(1)
    servo.duty_u16(MAX) #Move to maximum

```

4

Our system is only going to be able to do a certain number of things, so we can model it as a finite state machine. The states could include:

- idle
- out of food
- awaiting new food
- bird detected
- squirrel detected
- feeding (the birds)
- scaring (the squirrels)

Final system



We could describe the system in pseudo-code as follows:

```
Initialize state to idle
Forever
      SWITCH state
CASE state = idle
      IF bird detected
      THEN state = bird detected
      IF squirrel detected
      THEN state = squirrel detected
      IF ToF = EMPTY DISTANCE
      THEN state = out of food
      CASE state = bird detected
            LOG bird detected
            SET open_time = NOW
            OPEN servo
            state = feeding
      CASE state = feeding
            IF now - open_time = FEEDTIME
            THEN CLOSE servo
            state = idle
      CASE state = squirrel detected
            LOG squirrel detected
            SET sound_time = NOW
            START buzzer buzzing
            state = scaring
      CASE state = scaring
            IF now - sound_time = SOUNDTIME
            THEN STOP buzzer buzzing
            state = idle
      CASE state = out of food
            LOG out of food
            SEND Push notice “OUT OF FOOD”
            state = awaiting new food
      CASE state = awaiting new food
            IF ToF < EMPTY DISTANCE
            THEN state = idle
      END SWITCH
```

By planning our system into sub-systems and modeling it as a finite state machine, the code becomes simpler to debug. The above code would also lend itself well to programming in multiple functions (which themselves could be split across multiple modules if relevant).

With the code planned, all that remains is to program it in Python and test it. There is also the little job of building the automatic feeder itself! Luckily, you probably have all the materials needed in your recycling bin: the feeder funnel could be the top half of a soft drink bottle, and a few layers of cardboard would form a good cap. Everything can be held together with hot glue and gaffer tape while you finesse the design.

Testing

You can test the system with pictures of squirrels and birds, but the real test is putting it in the wild. Position it somewhere you can see it from a window and monitor what happens when the feeder is visited. Keep a log in a book and compare this to the log files you write to the SD card as the states change. Explore how long to leave the servo open to release just the right amount of bird food.

Stretch tasks

- At a minimum, log when either birds or squirrels are detected. Use Jupyter Notebook to analyze how long the birds and squirrels stay on the feeder. You may also be able to tell whether your device is scaring squirrels off, or teaching them not to try to steal food at all. This could be written to a CSV file and labeled by the human observer.
- There are other (non-AI) cameras that can be triggered via the Pico W. Explore these options and try to find one that will let you take high-resolution images of the birds visiting. Once you are able to identify which birds visit and when, correlate the data to your feeding records and try to identify the feeding habits of different species.
- Enable a simple web server on the Pico W so that you can get a real-time understanding of how much food is left in the feeder.

Final thoughts

This book has introduced a number of ways to use technology. While the projects require technology in order to be carried out, they have attempted to explore opportunities that have some sort of positive impact. This final chapter is no different and we would encourage you to look at what other waste can be reused, with the aid of microcontrollers such as the Raspberry Pi Pico W. The additional camera, for example, could be repurposed from an out-of-date digital camera with the shutter triggered from the Pico W. The buzzer or piezo could be recycled from old/broken toys, and another butter tub could form the body of the waterproof container. However you approach these projects, reducing waste, and in particular e-waste from unwanted electronics (which often contains amounts of various rare, and often toxic, metals) should always be a factor in your considerations. The project you have built serves one problem, but using outside-the-box thinking to come up with innovative solutions to a problem is what will drive the possible attainment of Global Goal 9, developing industry and infrastructure, while still protecting and rescuing our planet.