CYBS
"Cybersecurity"

Practical Work 1: Introduction to the Internet of Things

The goal of this practical session is to familiarise yourselves with IoT devices. You will firstly interact directly with the microcontroller itself, to understand how these devices work and what you can do with them. Then, you will develop your first bit of code and upload it to the device to run automatically. Finally, you will be confronted with various challenges, where you will need to develop your own code to reach the objective.

For this session, you will be using a Raspberry Pi Pico W[1]. Contrary to its bigger brothers, the standard Pi and the Pi Zero, the Pi Pico does not run with an Operating System, simply interpreting and executing the code its provided, being the Raspberry's first MicroController. This does provide some advantages, such as no configuration of the OS is needed, as well as much lower expectations on hardware and specs. Table 1 presents the specs of this device. You can compare them to the specs of the Raspberry Pi 5[2] discussed during the earlier lessons.

Table 1: Raspberry Pi Pico W – Specs

Chipset	RP2040[3]
CPU	133 MHz dual-code 32-bit ARM Cortex M0+
RAM	264kB SRAM
Storage	2MB on-board flash
Connections	1 x USB 1.1 2 × SPI 2 × I2C 2 × UART 3 × 12-bit ADC 16 × controllable PWM channels 26 × multi-function GPIO pins
Power	38mA to 72mA @ 5.5V via GPIO - down to 16mA in deep sleep Can be powered by USB
Networking	2.4GHz 802.11n WiFi Bluetooth 5.2 / Bluetooth Low Energy (BLE)

Preparation of the Pi Pico W

Before we can start, you need to familiarise yourselves with your little Pico. Figure 1 shows the layout of the board. As with many MicroControllers, the Pi Pico possesses 26 *General Purpose Input/Output* pins, otherwise known as GPIO. Each GPIO pin provides a certain functionality, allowing for example to attach and power an LED, or to extract data from an external thermometer. It is to be noted that not all pins are part of the GPIO, and provide other functionalities, such as power input/output or grounding for external modules. Take a moment to observe the board and identify the different elements, such as the placement of GPIO pins `1` and `26`. Note in particular that although these numbers don't correspond to the identification numbers of the pins in Figure 1, Raspberry have very kindly imprinted them on the underside. Some GPIO pins, however, are kept reserved by the controller itself and not exposed through the outside connectors. This is the case of `WL_GPIO0` which is connected to the onboard LED. The Pico also possesses other internal GPIO pins which are in use by the chipset, allowing for example to check the onboard voltage.

[1] https://www.raspberrypi.com/documentation/microcontrollers/raspberry-pi-pico.html
[2] https://www.raspberrypi.com/documentation/computers/raspberry-pi-5.html
[3] https://www.raspberrypi.com/documentation/microcontrollers/rp2040.html

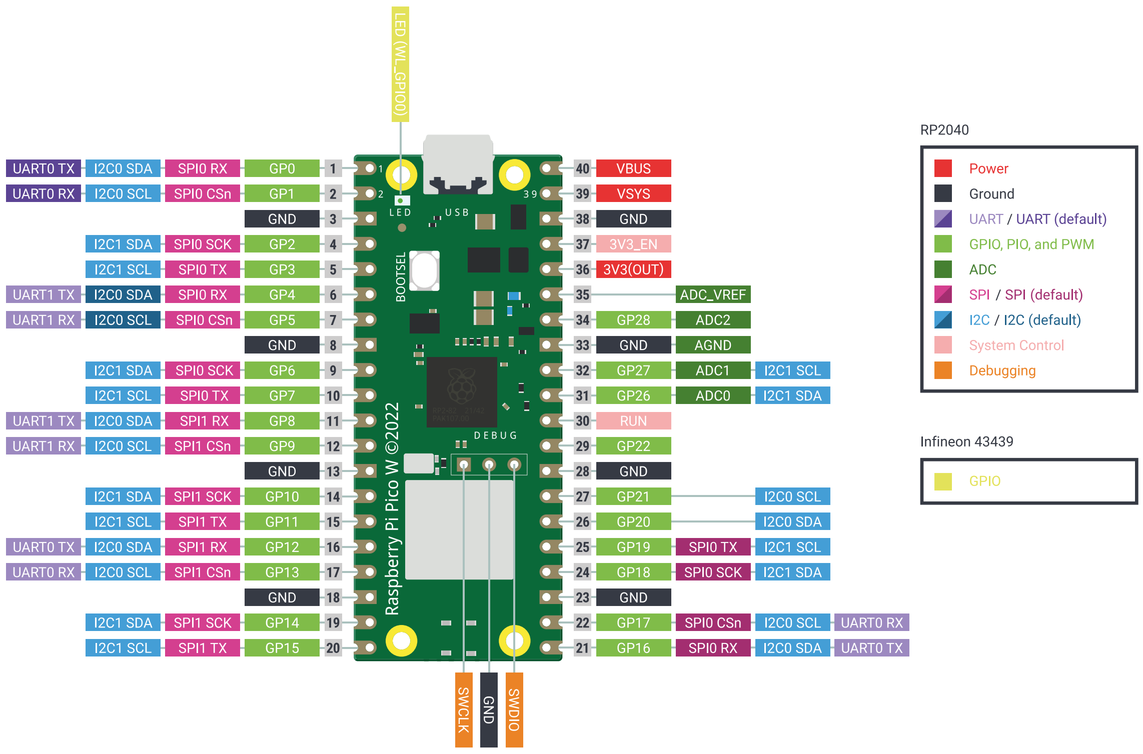

Figure 1: The Pico W layout

The Raspberry Pico family, although small is actually quite versatile. Indeed, there are currently two Pico's available for use: the "plain" *Pico*; and the *Pico W*. Although generally identical in their functionalities, the *Pico W* provides a much needed advantage in IoT: networking. As you may have guessed, identified by the addition of *W* standing for "wireless", the Pico W possesses an onboard wireless antenna. This antenna provides the ability to utilise two wireless network protocols: WiFi and Bluetooth. With this comes certain specifics, such as extra internal GPIO pins to operate and monitor the wireless module and antenna.

For this course, you will be using the *Pico W* as you will need WiFi to achieve your goals. Take a moment to observe your board and identify both the `WL_GPIO0` LED as well as the onboard Antenna. Verify that the board you have is indeed a Pico W, in case a normal Pico has slipped through.

Programming on MicroControllers is possible using different programming languages. For example, Arduino's generally utilise a derivative of C++, where as the ESP32's utilised in the security demo generally revolve around C. That being said, these aren't the only limitations as it is possible to implement your project through Java, Rust or even directly with assembly. However, here we will not discuss these languages and still to one you all know: Python.

Many devices, including both Arduino and ESP32's support programming with Python. However, there are certain limitations. As you know, compared to other languages such as C or C++, Python is an interpreted language, not needing compilation. This means that whereas other languages are converted to assembly and byte code for execution directly by the onboard processor, Python needs its own interpretation library on the device in question (this equates to installing Python3 on your computer before being able to execute a python script). Since these devices possess limited storage capacities, a slimmed down version of Python have been created called *MicroPython*[4]. Very close to the normal python, certain functionalities have been either reduced or removed to make space on the device[5].

[4]https://docs.micropython.org/en/latest/
[5]https://docs.micropython.org/en/latest/differences/python_310.html

Installation

Before we can start, we need to flash the Pico with a binary containing the MicroPython library and interpreter. To do this, download the firmware file `rp2-pico-w-firmware.utf2` from Moodle to your machine. Now, follow these steps to "flash" your Pico.

1. Connect the USB cable to the Pico **without** connecting it to the computer.
2. Hold down the `BOOTSEL` button on the Pico and connect the USB to the computer. Once connected, you can release the button.
3. The Pico will mount as an external drive called `RPI-RP2`. Open the drive and drag and drop the **UTF** file from Moodle onto the drive.
4. The Pico will disconnect and restart.

Your Pico is now ready and running MicroPython.

Configuration of VS Code

To implement your code, you will once again be using VS Code. However, although VS Code is capable of understanding Python and can help you with your programming syntax, natively it cannot communicate with MicroControllers without a little help.

Extensions are a major part of VS Code and are very useful. Here we will install some extensions to allow VS Code to communicate directly with your Pico. Make sure to close VS Code before proceeding to avoid conflicts.
Open a command line and type the following:

```
$ code --install-extension paullober.pico-w-go
```

Once installed, open VS Code once more and open a new folder on your machine. Open the VS Code command pallet using `Ctrl+Shift+P` (`Cmd+Shift+P` if on a Mac) then select `Pico-W-Go > Configure Project`. Your workspace is not setup and ready to be used. To connect to your Pico, check the blue toolbar at the bottom of the screen and click on the "*Pico Disconnected*" button. It should switch to "*Pico Connected*" indicating that the serial connection has been achieved.

Testing the Pico connection can be done by clicking the *Run* button at the bottom of the screen and going to the *Terminal* tab. You should see a standard Python terminal output, if not, check the terminal version is `Pico (W) vREPL` on the right. Test the connection with a simple `print("Hello World!")`. You should see the text output to the terminal.

You are now ready to start coding!

1 Getting to grips with the Pico W

Interacting with the Pico

For the first few exercises, you will be executing code directly on the Pico through the terminal you just used.

Many microcontrollers possess different onboard sensors or actuators. This is the case of the Pico, which possesses one of each: an internal temperature sensor and an LED as previously seen. Many others can be added via the GPIO, as demonstrated with the demo during the course Firstly, we are going to interact with both the onboard LED and temperature sensor.

LED

To interact with GPIO pins, internal or external, Python provides the `Pin` class through the `machine` module. Start by importing the `Pin` class into your terminal. Pay attention to the use of capital letters! The `Pin` classes constructor takes multiple variables, however, here we only need to use two: the `id` and the `mode`. The `id` corresponds to the GPIO pin, as presented in 1. The `mode` represents the type of device connected to the pin and how the Pico should interact with it. Two main modes are generally utilised: `Pin.IN` representing an input (sensor), and `Pin.OUT` representing an output (actuator). What is the GPIO `id` for the onboard LED and, in your opinion, what mode should you use? For help, Raspberry have kindly added a shortcut id for the onboard LED, as `"LED"`. The '"' are necessary in the command also.

Initialise the LED device using the `PIN` class as follows, replacing `<<id>>` and `<<mode>>` with the corresponding `id` and `mode`:

```
>> led = Pin(<<id>>, <<mode>>)
```

To modify the LED value, you can simply use the `led.on()` and `led.off()` functions to turn it on and off. Try them now. You can also utilise the method `led.value(<<state>>)` where `state = 0` turns the LED off and `state = 1` does the opposite. The `led.value()` function, when called without a parameter, returns the current state of the LED, following the previous principal (0 = OFF, 1 = ON).

Temperature

As stated previously, the Pico possesses an onboard temperature sensor which we can read. However, unlike the LED we we cannot simply access the digital information as this sensor is analogue. Indeed, only 23 of the 26 GPIO pins are digital, leaving the other 3 capable of conveying and understanding analogue information. To access the information, we need to use an *Analogue-to-Digital Converter*, otherwise known as an *ADC*. Try and identify the three *ADC* pins presented in 1. You should also see the presence of another, not associated with an unused GPIO pin.

In total, the Pico provides five *ADC* channels, one of which, `ADC 4`, is connected to the internal temperature sensor. To access it, we need to utilise the `ADC` class of the `machine` module. Once imported, create an instance of the sensor as follows:

```
>> sensor = ADC(<<channel>>)
```

To utilise the information taken by the sensor, we need to firstly convert it into a digital value. We then need to convert the information to a temperature value we can use.

```
>> conversion = 3.3 / 65535
>> reading = sensor.read_u16() * conversion
```

From now, we can convert the value to the actual temperature. You may have noticed we haven't called this sensor a "thermometer" at all. This is because it doesn't sense the temperature itself, but the voltage of an onboard sensor (a biased bipolar diode). From the documentation, we know that the voltage of the sensor at 27 degrees C is 0.706V, with a slope of -1.721mV (0.001721) per degree. Thus, we can calculate the temperature using the following :

```
>> temp = 27 - (reading - 0.706)/0.001721
```

Print the temperature of your board.

Pushing Code

Until now you have been running your code manually on the Pico through the serial connection on the terminal. Although this is OK for our use case, it does have some limitations. To illustrate this, write a simple `while True` loop to change the `LED` status every second. You can use the function `led.toggle()` to change the `LED`. Don't forget to import the `time` library as well as the correct code indentation. What do you see when your code is running?

The Pico is now stuck, running your code and you can no longer access it. This is normal as the `while True` loop is running indefinitely. Disconnect and reconnect the Pico's USB cable to reset it then reconnect to your Pico and press *Run* to reset the terminal. On MicroPython devices, code structure is different to that of normal Python code. MicroControllers generally have two Python files containing the code: a `boot.py` file, containing code which is run as soon as the device starts; and a `main.py` file, which contains the general code. In practice, code is generally written in `main.py`, leaving `boot.py` for the setting up of other elements, such as networking for example.

Create the `main.py` file, recreate your LED loop in it and run the code on the Pico by pressing *Run* at the bottom. What do you see? Try and stop the code and see what happens.

Now modify your loop to print the temperature instead of toggling the LED. To make it easier, create a function `def getTemp(reading)` that takes the reading from the sensor, converts the value to digital before returning the calculated temperature. Your while loop should look like this:

```
while True :
    time.sleep(1)
    reading = sensor.read_u16()
    temp = getTemp(reading) # Your code runs in this function
    print(temp)
```

Listing 1: Temperature main loop

Now, update your code so that your `while True` loop either toggles the LED, or reads the temperature based upon the value of a boolean variable `LED`. Set `LED = True` and run your code. You should only have the LED flashing on your board. Now move the `LED = True` to a new file, called *boot.py*. What happens now if you run the *main.py*? How can you correct the error you get? Try changing the value of `LED` and see if your code reacts accordingly. Your main loop should resemble this:

```
while True :
    time.sleep(1)
    if LED :
        led.toggle()
    else :
        reading = sensor.read_u16()
        temp = getTemp(reading)
        print(temp)
```

Listing 2: Main loop with variable separation

To finish, set `LED = True`, then instead of running your code, right click on both files (*boot.py* and *main.py*) and select *"Upload current file to Pico"*. Click on the button *Toggle Pico-W-FS* at the bottom of the screen and check if both files are on your device. Now close VS Code, then disconnect and reconnect your Pico. What do you see? Try connecting it to an external battery the professor has.

2 Wireless and Web-ing

Now that you have managed to upload and run autonomous code on your device. However, as of yet we still cannot interact and request data from the sensors on your Pico. To do this, we will use a WiFi network with a web interface. Here you will recreate a simpler version of the demo performed during the course, once again visible at the front of the room. Firstly, utilise the *Toggle Pico-W-FS* button and remove both files from the *Pico W Remote Workspace* using right click.

Connecting other devices

Download the *boot.py* and *main.py* files from moodle and add them to your workspace. Take a look at them to understand what they do. In the *boot.py* file, choose an appropriate `SSID` and `PASSWORD` for your Pico. Take care to put a simple password here as you will need to use it to connect to the device either from the computer or your phone. Run the code and check if the network has been started from the terminal. You can also connect to your device using your phone or computer. Run the *main.py* file and try to connect to the IP written on the terminal using a browser. You should be met with a welcome message.

Now open the *main.py* file and find where the website data is stored. Create a variable `NAME` at the top of your file which will contain a custom name for your Pico. Modify the HTML data to complete the *"My name is ..."* sentence as well as the code further down to include your name on the webpage. You are now ready to interact directly with the LED and temperature sensor on your device.

Interacting with your Pico

Scroll through *main.py* until you find the sections marked `TODO`. Read the associated comments and create the necessary code to allow the website to print three things:

- **The devices uptime**. Utilise the time library to get the milliseconds since the device was powered up. Utilise the time conversion function at hand to print it in an understandable format.
- **The temperature**. Utilise the code and functions you created before to add the temperature into the website when requested.
- **The LED**. Find where the website interacts with the LED and add the code to change its status.8956

Check each stage of your work through the browser. Your output terminal also contains information to guide you in your work. Don't forget, that after each modification you need to click on the *stop* and *run* buttons at the bottom of the page.

When you have created your website, call the professor over to check it out. When all is clear, you can move onto the next exercise.

3 Challenge-based programming

The goal of this exercise is to confront you with multiple challenges you must overcome. There are a total of three levels to beat, each with 100 challenges, and each harder than the last. The goal is not to complete them all, simply to challenge yourselves to resolve the problems algorithmically.

Environment setup

The challenge will proceed as follows:

1. you will connect to the level you are on
2. you will receive a task to achieve via message
3. you will have a limited duration to complete the task, generally under 1s (except specific cases)
4. you will reply with the answer
5. if you were correct, you will move on to the next challenge. If not, you will receive a failure message and have to start the level again.

Although this sounds daunting, the challenges will remain generally the same, with only the contents of the tasks being random. This means that if you reached challenge 60 of level 1 and fail on challenge 61, you have already completed the other 60, so you only need to resolve challenge 61 to move forwards.

As stated, you are not expected nor required to complete all levels 100%, simply applying algorithmic thinking to specific tasks. To help with this and allow you to concentrate on your algorithms, the source code has been provided on moodle. You only need to modify the header of `main.py` to change the network ssid, password and add your unique group ID, which is explained below. Your main interest will be on the `def challenges(task)` function.

Your challenge ...

Each task will be passed to the `challenges` function that you will be extending. The task parameter represents a dictionary, with at least one key: *type*. This entry corresponds to the task you need to perform. This will allow you to differentiate between tasks using `if ... elif ... else`. Three task types have already been provided in the `challenges` function: a *SUCCESS* task, to inform you when a level has been completed, a *FAIL* task, to print the correct and incorrect answers, and a *STARTUP* task, to confirm your device is ready and start the challenges. There is also an `else` clause which returns `UNKNOWN` when an unknown challenge arrives.

As you can see, all three types have different keys in the dictionary. The dictionary structure is specific to the task at hand and is explained with each task information. Once you complete one level, you will be able to move on to the next level.

May the best group win!

As stated, there are three levels, all running on ESP 32's, named "*CYBS_CHALLENGE_1*", "*CYBS_CHALLENGE_2*" and "*CYBS_CHALLENGE_3*". Each of these networks are password protected and you will need to beat the previous level to receive the password for the next. You will receive the password to the first level when you have completed the previous exercise. Before starting, you must select a group id, which you will need to put in your `main.py` to identify you.

On the board will be projected the overall progress of each group. Good luck to you all!