Exploring the Espressif ESP8266 and ESP32 with Micropython

Science Oxford Python in Education Conference 2024

John Gouk (johngouk@gmail.com)

Introduction

The ESP32 and ESP8266* chips from Espressif are WiFi-enabled, and in the case of ESP32, Bluetooth-enabled SOCs (Systems on a chip).

- They come with enough RAM and Flash (non-volatile RAM) to run the Micropython port of Python, which is more than enough to do some amazing things, especially when they are connected to the Internet and can send and receive data.
- They are ideal components to enable the Internet of Things - their GPIO ports allow connection of pretty much all the standard sensors and displays, and they are powerful enough to support a simple webserver for configuration or data display.

I have instrumented my Daikin Heat Pump with an ESP32, and it has provided great feedback on its configuration, as well as heat generated vs. energy consumed.

* I use the terms "ESP32" and "ESP8266" fairly loosely in this discussion; Espressif supports a myriad of versions of each family e.g. the latest ESP8xxx is ESP8684, but the general points apply; also, stocks of chips of all eras are available

Note:

- **There are lots of different SoCs available, RP2040, RPi Pico, etc. I'm talking about ESPs because I've used them!**
- **This is intended to be a lightweight intro to MicroPython and ESP-land; I am not an expert, and if I get stuck, I use C++...**

MicroPython vs. CPython, ESP vs. micro:bit

If you want to teach programming using "Python", then I would suggest you stick with CPython, or other similar full Python implementations, running on a suitable desk/laptop or PythonAnywhere. There's no need to introduce the complexity of getting ESPs and the ESP support infrastructure organised to learn programming.

If, on the other hand, you have some more general, wider STEM-based requirements, then using microcontrollers or SOCs is a great way to get physical, and grasp the underlying principles of how the 21st century actually works:

21st century design is the choice of the appropriate computer, writing software and then fiddling with some attached clanky bits

John Gouk, 2022

There are lots of choices of MCUs and SOCs – here's an utterly superficial comparison of ESPs and μbits:

Topic	Microbit	ESP
Cost	Basic unit more than ESP (£13) Only one provider, so limited choice (1!)	Free market madness… eBay prices today: £4 Wemos D1 Mini ESP8266 £6 ESP32 Aliexpress: £1.50 ESP8266 £3 ESP32
Display	5x5 LEDs – basic!	None, but anything goes
Pins	Lots	Enough or lots (8266 vs 32)
Speed	Meh	OK/Wow
Cores	1	1 or 2 (8266 vs 32)
Sensors	6*	None, but sensors are cheap, and your own choice
Flash/RAM	Not much! Can't	Some or Loads; makes Python possible
Connectivity	Radio, kinda BLE	Wifi, both STA and AP! BLE! ESP-NOW (mesh)
Potential users	KS2 – KS2.5/3	KS3 onwards
Programming	Blocks, MPy, JS C/C++ possible but…	MPy, C/C++, LUA, others Lots of OS efforts out there
Resources, examples, libs	Lots; if from microbit.org then probably reliable	Lots, may require tweaking, but then, you're learning…

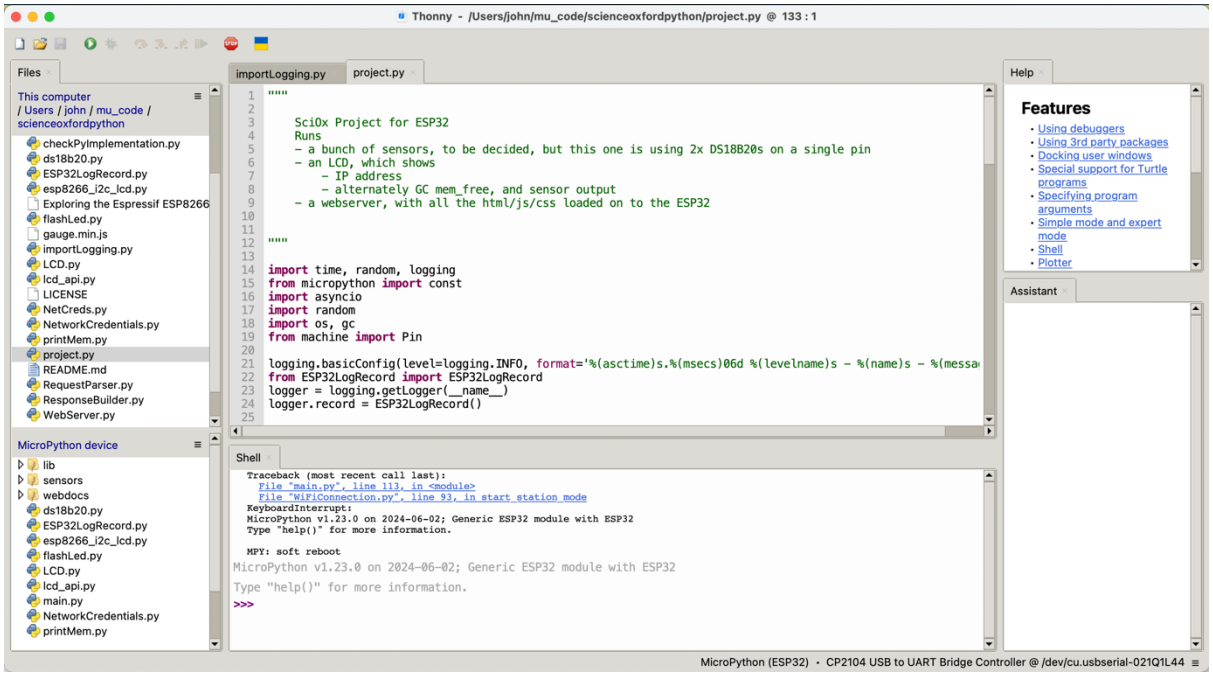
* Accelerometer, Magnetometer/compass, Temperature, [V2: Mic, Speaker], 2 buttons

I've used microbits in Code Clubs with Y6/7/8, and they love them. I haven't done anything with ESPs yet as part of a Code Club, although Sarah Townson and I have plans for SciOx CC next year… As ever, it depends what you want to do. If you want to design something with no extraneous components, only the ones you want for a specific purpose, that has proper outside-world connectivity, the ESP family is the way to go. Its flexibility is its key, and getting data on and off it is easy, which is perfect for data gathering, IoT and so on.

Choice of IDE

There are many IDEs (Integrated Development Environment) that enable development in Python on SOCs, like ESPs, microbits and so on. If you have a favourite Python IDE, like Mu, it may well already support SOC development – Mu does, for example. I am currently using Thonny, largely because it supports

- Viewing/manipulating SOC and computer filesystems at the same time, including easy uploads/downloads to/from SOC/ESP
- Editing multiple files simultaneously, on both computer and SOC (super handy!!)
- A version of Python REPL, which is great – try out the commands on ESP32 instantly!



The screenshot shows the main features:

- File system windows on the LH side
- Editing multiple files in central window
- Shell window at the bottom
- Help and Assistant windows on the RH side – you can close these to increase useful windows, they are really only useful with CPython, which Thonny also supports well
- At the very bottom RH side, you can see the current environment choice and USB connection to the ESP board

ESP Development Boards

Although ESPs are available as bare chips, this is a pretty useless object for mere mortals. You're better to get a development board, which includes

- a serial-USB interface to communicate with the chip (very handy!)
- easily accessible sockets or solder points exposing the various hardware ports/pins
- a power supply point, usually combined with the USB socket to take 5V from the USB

These are easily found on Amazon (if you don't mind Jeff's gross self-indulgence at your expense), AliExpress (slow but super cheap), EBay and other outlets. I've just done a simple search for "ESP32 development board" on these sites to get these links.

ESP8266 vs ESP32

The ESP8266EX (or its predecessors) was the original Espressif WiFi-enabled SoC introduced in 2013. It has a single core, 160Kb of RAM (around 50Kb left when wifi-connected in Station mode), maybe 4Mb of Flash, 1.5 UARTs for serial comms, and 17 GPIO (General Purpose Input Output) pins.

Its big brother, the ESP32 family, has two cores, 520Kb of on-chip RAM, a variable amount of external Flash (16Mb anyone??), 2 UARTs, and 32 GPIO pins. It also has a low-power coprocessor, which allows for a very low power sleep mode – handy if you're a battery powered device! It also has SPI and I2C in hardware, which is very handy for connecting smart sensors...

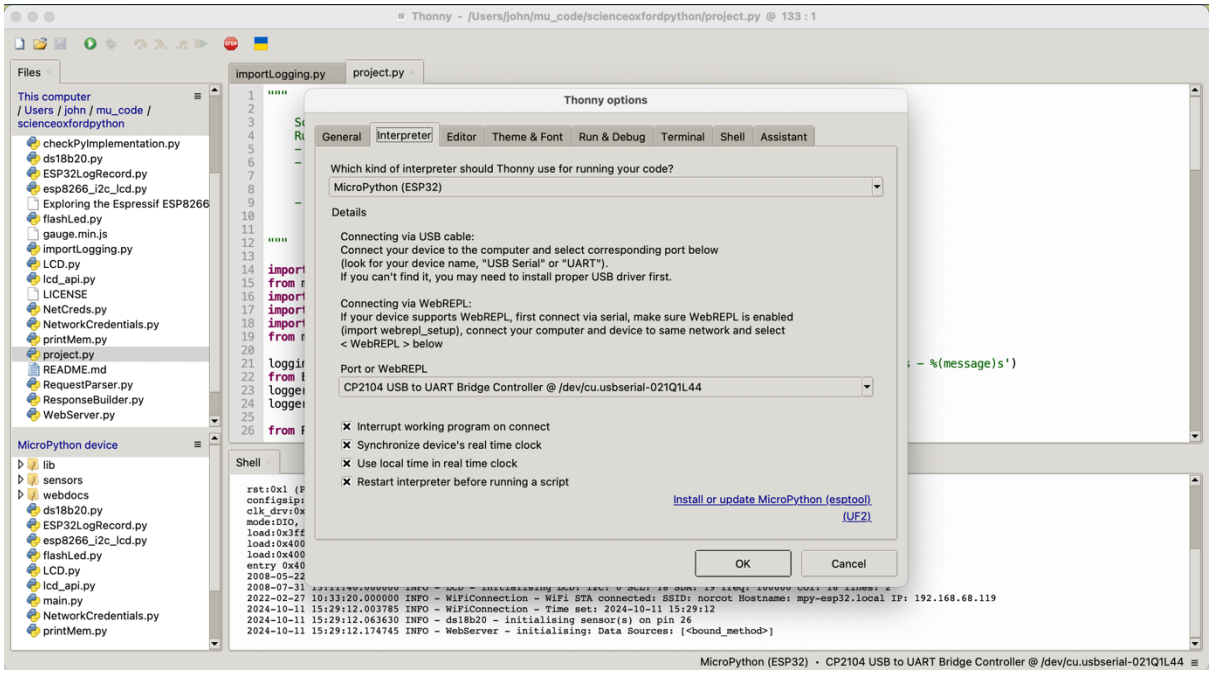
If you don't have a power supply problem (like not enough!), I'd go with the ESP32, on the basis that you can run much bigger Python programs on it, and it doesn't cost any more in the kind of usable packages available.

Connecting ESPs to your computer

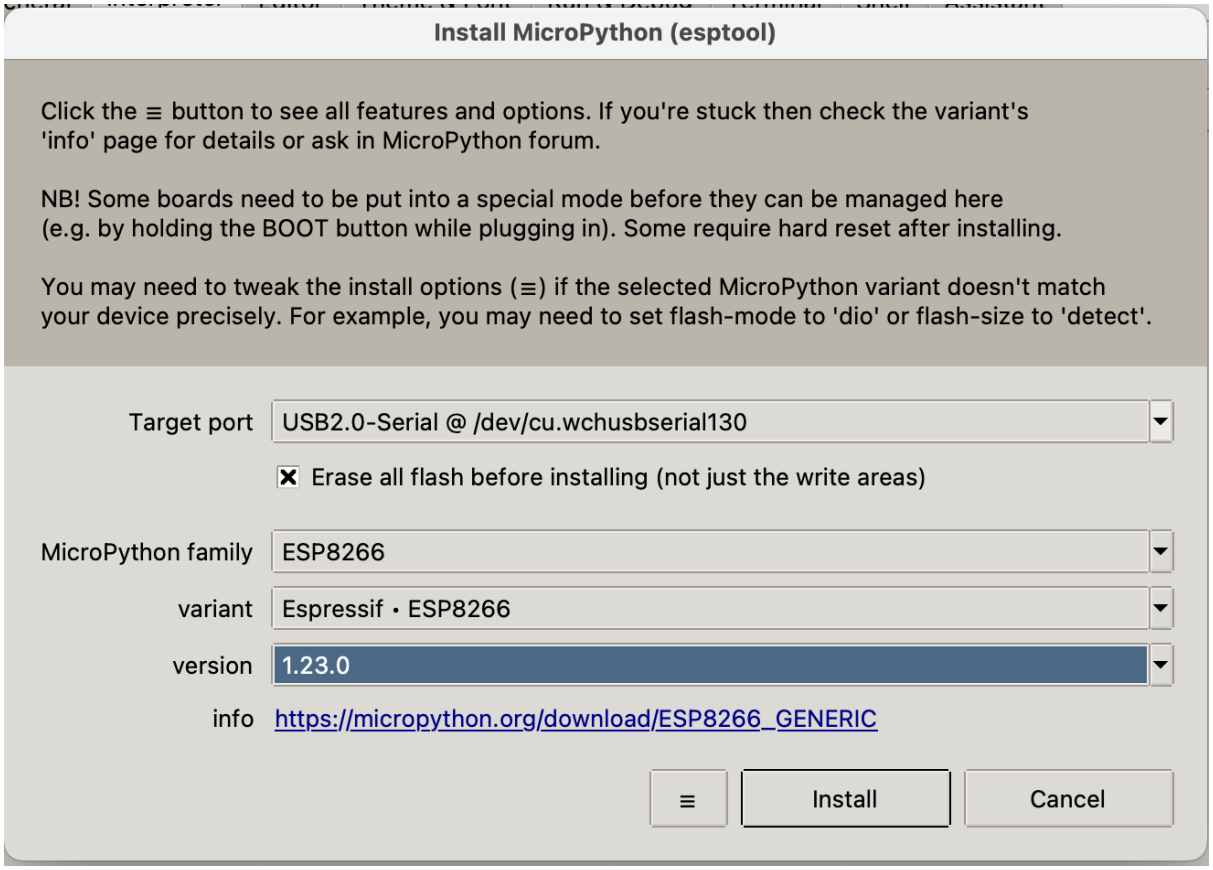
For the physical connection, this is in principle pretty simple – use an appropriate USB data cable to connect the dev board USB socket to your computer USB. The only potential issue is USB drivers – some USB UARTs (the bit that allows the dev board to talk USB) require updated or different drivers, especially the WCH family. If this happens, check out their driver download page – you are looking for something like CHxxxSER.ZIP[_MAC|_LINUX|.[EXE|ZIP], depending on your OS and the chip type. Give it a go before you try to fix it!

Now you need to connect your IDE to the new USB/serial port. Open the IDE of choice – my example will be Thonny, but Mu and others work the same kind of way. Use Thonny/Preferences and the Interpreter tab – see below. Select the ESP32 or ESP8266 interpreter – Thonny starts with CPython if you want to try it with that, running some scripts on your computer first. Select the port the ESP is connected to – that's the hardest part sometimes! Look in Device Manager USB on Windows to see what's

connected, if nothing like a serial device, then maybe you need additional drivers.



Since this is the first time, click on “Install or update MicroPython (esptool)”. This will install the MPy interpreter on the ESP, which is required to make this all work. Use the pop-up window to configure the right device for the MPy version:



Select the same port as the previous panel, the ESPxx(xx) you are using, and if an ESP32, the correct variant. Use the most recent stable version (1.23 at doc time). Press Install… a progress bar and status messages appear to the left as the Espressif Python-based esptool writes the new code to the ESP. Done! You're ready to go.

MicroPython Docs

You'll need the documentation at https://docs.micropython.org/en/latest/index.html . At some point, you'll want to read the standard Python lib info, the MPy lib info, and the ESP32/ESP8266 port-specific info to make progress beyond the simplest things, especially if you want to use chip-specific features. I just read what I needed when I needed it… MPy is cool, but the limited space on the chips means lots of the more esoteric Python libraries didn't make the cut… but that doesn't prevent lots of interesting possibilities.

Demos

I've set up some sample and demo code in https://github.com/johngouk/scienceoxfordpython, which I'll be talking through on the walk-up session. The items are intended to show some of the most interesting, useful or tricky features of the ESP I've encountered recently – I'm sure there are many more!

WiFi Connections

STAtion Connect

Connecting to wifi with an ESP is incredibly simple, as MPy provides a library `network` that does it all. `simpleConnect.py` illustrates this (replace ssid and password with appropriate values):

```
import network
# Connection
w = network.WLAN(network.STA_IF)
w.active(True)
w.connect("ssid", "password")
while not (w.isconnected()):
    pass
ipaddr = w.ifconfig()[0]
print('Connected! IP: ' + ipaddr)   # You're connected!
print('Hostname:', network.hostname())
print('Available networks:',w.scan())
```

It's as simple as that. The while loop is to allow the ESP to connect – this can take some seconds. Now you can ping the ESP from a terminal prompt on your computer at the IP

address or using the hostname + ".local" – the ESP RTOS and Python implement the mDNS functions when you connect.

Access Point (AP) Create

This is just as simple (`simpleAPMode.py`), in fact more so, because you don't need a separate WiFi network, the ESP starts its own Access Point, which you can join with your laptop or phone – the ESP will print its SSID and the password on the IDE shell window.

```
# AP Mode example - try accessing the AP using the SSID!
import network
ap = network.WLAN(network.AP_IF)
ap.active(True)
# Next required to allow iPhones etc. think it's a network ☺
ap.config(authmode=network.AUTH_WPA_WPA2_PSK,password=password)
ap_ip = ap.ifconfig()[0]
ap_ssid = ap.config('ssid')
print("SSID: %s Password:'%s' IP:%s"%(ap_ssid,password,ap_ip))
```

Both AP and STA?!

Yes, you can actually run both at the same time. It's relatively easy to try the STA for the specified SSID/Password, and if those don't work, start an AP to which the user connects and then uses a web page to reconfigure the network access credentials. But then you'd need a web server, right?

Loading Optional Modules

Since the ESPs have very little storage and RAM, MPy has been compiled without a lot of the standard Python libs. However, these are available if you want to use them, and the `mip` tool allows them to be downloaded onto the ESP file system when you need them. It's almost magic… Just set up a wifi connection, then invoke mip…

```
import network, mip
# Connection
w = network.WLAN(network.STA_IF)
w.active(True)
w.connect("ssid", "password")
while not (w.isconnected()):
    pass
ipaddr = w.ifconfig()[0]
print('Connected! IP: ' + ipaddr)   # Need this to connect!!

mip.install('logging')
mip.install('time')
```

It's as simple as that. You can then

- see a lib/ directory on the ESP filesystem in Thonny
- import the libs you loaded for use in your code or the REPL

Logging data to files e.g. CSV

`logDataToCSV.py` demonstrates the ease of logging data to files on the ESP file system. It's not really a lot different from CPython, to be honest. The only wrinkly bit is the time handling and formatting – ESP MPy doesn't do Epoch time as a `float` of `ssssssss.mmmiii`, it's a nominal `int` of the form `sssssssssmmmiiinnn` where the `nnn` nanosecond component is always `000`. Doing the nanonseconds to seconds etc. with FP arithmetic breaks it all, so I (horribly) convert the 18 digit integer to a string, and convert the right bits back to `int`. If you can think of a better way, let me know!

We'll look at downloading the generated data files using a web server on the ESP later.

Logging for errors, info, debug etc.

I don't know about you, but I like programmes to tell me what's happening, at an appropriate level for the current stage of development – super verbose when you start, dire errors or warnings only when you're in production. CPython does this with the `logging` module, which allows for runtime configuration of the level of logging, even by module if you want. Logging can go to files, the console (screen), or anywhere you want to write a simple connector for.

MicroPython can do this too, in a restricted fashion. Try `basicLogging.py`, which demonstrates sending the same set of DEBUG, INFO and ERROR messages with different configured logging levels. There is no timestamp on the format used for these, which is not very handy… and can be adjusted by varying the `logging.basicConfig(format="…")` command.

However, MPy logging is broken on the ESP family, because their version of `time.time()` returns a huge integer for current time in seconds, rather than a decimal float with seconds.fractional seconds. I wrote a replacement `LogRecord` class that subclasses the original `LogRecord`, which when inserted into the `logger` object, properly records the values. You also have to import the `micropython-lib` version of `time.py` (see above). See `betterLogging.py` for an example. You'll see that it's showing GMT/UTC, not BST – see below for more on network time.

Logging to files on the ESP is possible, but I haven't made it work yet. You can just use open, write etc calls etc. to write files, including CSVs for data. These could be downloaded using a web server on the ESP…

Network Time

The ESPs have a not very good Real Time Clock (RTC), but you can at least set it to current GMT if you have a network connection, using the `ntptime` module.

```
import ntptime
ntptime.settime()
```

Then the `time.time()`, `time.gmtime()` and `ntptime.settime()` functions provide (kind of) realtime time, although since MPy doesn't support timezones, you'll have to sort that for yourself! Real programmers use UTC anyway, to prevent clock switches messing up time order...

And if you really want `datetime`, you can import it using `mip` (see above). I haven't tried it.

Checking the Python Implementation and Platform

You can write multi-platform code that runs on anything, provided you

- Stick to the common subset of libs
- Check where you're running and take appropriate action!

See checkPyImplementation.py – it uses `os.uname` and `sys.implementation.name` to check the hardware and Python type.

LCD Fun

Displays are cool, and the easiest is the classic 16x2 backlit LCD, preferably with an I2C backpack to simplify connection and use the least GPIO pins. There are lots of implementations, the demo just shows the possibilities. Check `simpleLCD.py` for a demo.

Asyncio – how to make lots of things happen "all at the same time"

The jewel in the crown of MPy, and probably CPy as well, is the `asyncio` lib. This allows one to write simple code that runs multiple tasks, each of which does something or starts something, then waits for a bit and comes back to do it again or check on the something's completion. This is called **co-operative multi-tasking**, and it's ideal for tasks that spend lots of time waiting for something to happen, rather than using the CPU.

For example, if you have a temperature sensor and an LCD, you might want to

- sample the temperature every 10 seconds, because more frequently doesn't make sense
- update the LCD with the time every second, or blink the colon in the time or something to show things are working

Now, you could write some very complicated loop that kept track of what it did last, constantly checked the current time since it last did one or other thing etc. etc., and spend a long time testing it. And it would be hard to change to implement a different set of requirements unless you put a lot of work into it. Believe me, I have done this!

Or you could use the `asyncio` module. This allows you to define a set of tasks, which are basically Python functions, that are passed control of the CPU by the central `asyncio` module, as and when they have asked to be. What they do how long they wait, and what they wait for is entirely up to them/you, but you don't have to put the underlying management together at all. Look at `asyncioDemo.py` for a simple CPython and MPy demo.

Web Server

Now we've introduced the `asyncio` module, we can look at the server object it supports, which is a complete web server service that can run alongside your own tasks. Use the web server to respond to data requests, and gather data from the sensors independently as and when you want. The `/WebServer` directory/folder has an example of an asyncio-based web server, which works for file serving, but not interactive requests.