

Lab8: ESP32 Board

Introduction

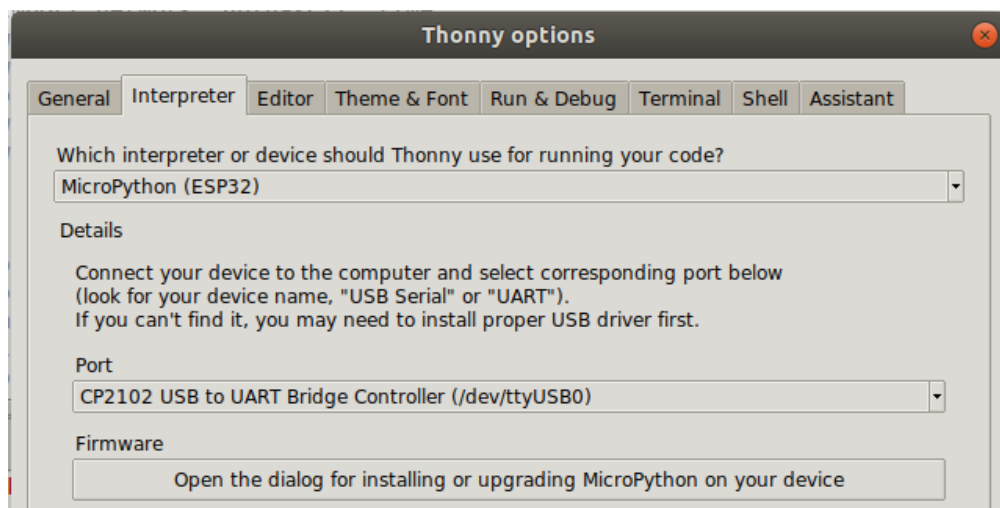
This lab introduces the ESP32 and the use of the Thonny development IDE. Several basic I/O scenarios are explored, including networking and the use of sockets.

Learning outcomes

1. Become familiar with ESP32 Board features and programming
2. Use the ESP32 to interface to simple devices (switches, LEDs, sensors)
3. Develop code to connect to the Lab network and then out onto the Internet

Task 1: Configuring Thonny for MicroPython

1. Start Ubuntu VM and log in.
2. Plug the ESP32 board into the USB port of the computer – when asked, select ‘Connect to Virtual machine’. If you do not get this invitation, go to ‘Player’ (top left of screen) and select ‘Removable Devices’. From there select the ESP32 (Disconnect from Host) and Connect to Virtual Machine. This may open up File Manager – just close it again.
3. For MicroPython programming, we have to change the interpreter from Python 3.7 to MicroPython. Run Thonny, go to Tools>Options... and select the ‘Interpreter’ tab.
4. Choose ‘MicroPython (ESP32)’, then select the Port as shown below which should appear in a drop down list:

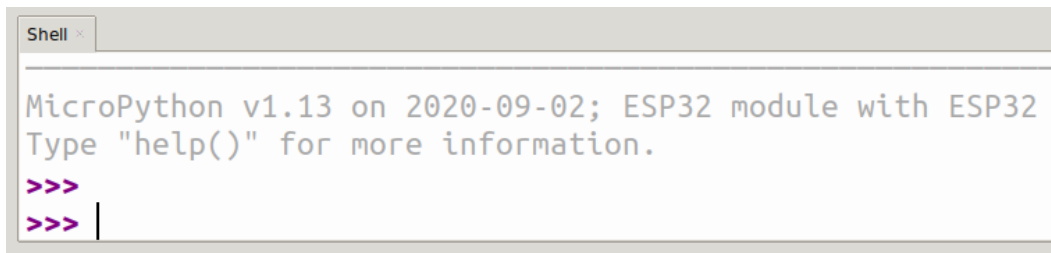


Click OK

5. At this point we want Ubuntu to connect to the ESP32 via a virtual Serial port. Most likely it will not, because you are not a member of the ‘serial port users group’ (called ‘dialout’).

Review your notes or look up on line how to make yourself a member of a group, then check using 'getent'. Finally, restart your machine for the setting to take effect.

6. Run Thonny again. If all is well, you will get a connection to the ESP32:



```
Shell x
MicroPython v1.13 on 2020-09-02; ESP32 module with ESP32
Type "help()" for more information.
>>>
>>> |
```

If you get messages in red, suggesting you restart the device, select Device>Close serial connection and then click the red menu icon Stop button (or Ctrl-F2)

7. Once you get the REPL >>> prompt, all is working. Note that the current version of MicroPython on the board is 1.13, and it is dated 2nd September 2020. Firmware updates containing the MicroPython interpreter are published from time to time and can be downloaded to the board.
8. Check correct operation by asking python to calculate 2^{100} (type `2**100` at the prompt, check the answer with your Windows Calculator)

Task 2: Basic I/O Programming

1. The ESP32 has an on-board blue LED connected to pin 2 on the ESP32 chip. It also has a Reset button (called 'EN') and a user input button (called IO0). Make sure you can differentiate between these buttons.
2. We will use the REPL to flash the ESP32 LED. At the prompt, type the following and note the result – watch out for errors which may be reported if you mis-type.

```
>>> from machine import Pin    # this loads a library module
>>> LED = Pin(2, Pin.OUT)      # Makes pin 2 an Output
                                # Pin 2 is connected to a blue LED
                                # refer to diagram on page 6

>>> LED.on()
>>> LED.off()
```

3. You will recall that one of the standard Python modules is a 'time' module, and it can be used to introduce delays into your program. Write a program (in the code editor this time) that will flash the blue LED ON for one second then OFF for 1 second. Make sure you save your program on your PC.

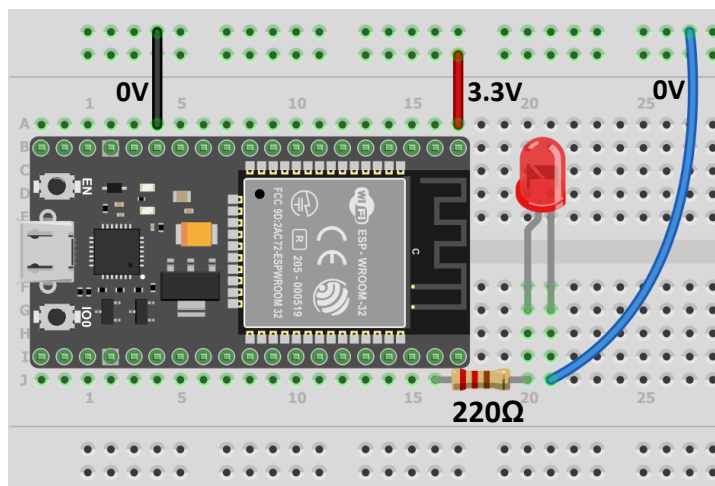
4. Add a while loop make the sequence repeat indefinitely. You can interrupt the execution of the while loop by pressing the reset button on the board.

At this stage you might want to start to refer to reference material on Python programming to check the syntax and usage of commands like 'while' and 'if . . else'.

5. The user input button is on Pin 0 (labelled 'IO0'). Write a program so that the LED flashes (as before) but stops when the user button is pressed down – this will take some thought. This is made more difficult because we have no 'debug' facility. Typing single line commands into the REPL will help you check the operation of the switch.

NOTE: Using 'print' statements to check program execution at critical points is the 'poor mans debugging system'. It can be quite helpful.

6. Instead of always calling a 1 second delay, replace the '1' with a variable, so that when the switch is pressed the sequence runs quickly, and when it is released it runs slowly.
7. We will now connect an external LED to the board and control it with a Python program. You need to get a red LED + 220 ohm resistor, along with a jumper wire (shown blue). Arrange them exactly as shown in the photo (enlarge on-screen if necessary):



- i) the (black) wire connects GND (check slide 5) to the blue strip of the prototyping board
- ii) the resistor connects from the pin GPIO23 to the 'anode' (long leg) of the LED
- iii) The LED connects from the resistor to ground via the jumper wire shown in blue (the longer lead must be on the left as viewed in the diagram, or it will not work)

Make sure you check your connections carefully before you proceed to apply power via the USB cable.

Check that the programs you wrote for the blue LED will also work for this external LED (you will need to edit pin configurations in your programs).

Task 3: Network Programming

1. Using the commands described in the lecture slides, write a short program to connect the ESP32 to your network and print out the IP address it has been given. This will be useful whenever you need to connect to the network. Code is given on the ESP page <http://docs.micropython.org/en/latest/esp32/quickref.html#networking>

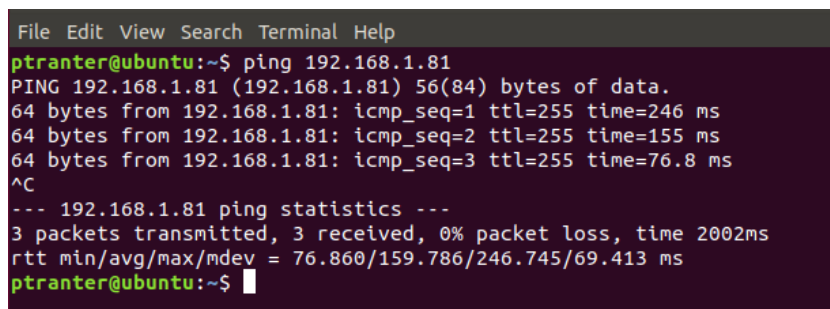
The code is given as a *'useful function for connecting to your WiFi'* with the name `'do_connect()'`

If you make a script file of this code, add your network name and password, and run it, you may be surprised to find that nothing happens. Is this an error? What you have is a **function** (it begins with the `'def'` keyword) not a program. Running it does load it into memory as a function that you may call, so if you then type `'do_connect()'` at the REPL, you will find that after a short wait you will get a message like this:

```
>>> do_connect()
I (1425021) phy: phy_version: 4180, cb3948e, Sep 12 2019, 16:39:13, 0, 0
connecting to network...
network config: ('192.168.1.81', '255.255.255.0', '192.168.1.254', '192.168.1.254')
>>> |
```

The first IP address (192.168.1.81) is the IP address allocated to my ESP32 on my network (yours will probably be different). At this point you have a connection to the network, and should be able to 'ping' your ESP32 from your Ubuntu VM (firewalls permitting!). Try it out.

Mine responded like this, confirming that 192.168.1.81 is valid 'ping-able' device address on my network



```
File Edit View Search Terminal Help
ptranter@ubuntu:~$ ping 192.168.1.81
PING 192.168.1.81 (192.168.1.81) 56(84) bytes of data.
64 bytes from 192.168.1.81: icmp_seq=1 ttl=255 time=246 ms
64 bytes from 192.168.1.81: icmp_seq=2 ttl=255 time=155 ms
64 bytes from 192.168.1.81: icmp_seq=3 ttl=255 time=76.8 ms
^C
--- 192.168.1.81 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2002ms
rtt min/avg/max/mdev = 76.860/159.786/246.745/69.413 ms
ptranter@ubuntu:~$
```

2. In lab 5 task 1 you prepared a program to connect to the internet and download a small web page. This needs rewriting for the ESP32.

MicroPython implements its own version of the Berkley BSD 'socket' object in a package called 'usocket' (most MicroPython modules have the same name as the Python modules, but add the letter 'u' to the beginning to denote MicroPython. It has less functionality than the full socket module, but is fine for our needs, and is smaller (uses less memory). The main change is line 1, which should be written to access the 'usocket' module. Note that doing `'import ... as'` allows us to change the name to 'socket' in our code

```

1 import usocket as socket # note we keep imported name as 'socket'
2 |
3 addr = socket.getaddrinfo('micropython.org', 80)[0][-1]
4 print(addr)
5 s = socket.socket()
6 s.connect(addr)
7 s.send(b'GET /ks/test.html HTTP/1.1\r\nHost: micropython.org\r\n\r\n')
8 data = s.recv(1000)
9 print(data)
10 s.close()

```

3. For this to work, you need to have run your 'connect' program first to connect to the network. Then run your web page program and note that the usocket module can read a web page URL and convert it to an IP address using the 'getaddrinfo()' function as before.

At the moment, we don't have a paho-mqtt library to run on the ESP32 – we will look at importing the module (called umqtt.simple) next week. If you look at the MicroPython Forum you will see that it has been the subject of a lot of posts.

4. We can use the ESP32 to get the correct time. The ESP32 have a real-time clock function, but obviously it needs to be told what the time is when it is powered up.

Write some code to get the time from a time server and print it out. The simplest one to use is at time-C.timefreq.bldrdoc.gov

The code to get the address (line 4) I slightly different:

```

1 import usocket as socket # note we keep imported name as 'socket'
2 |
3 s = socket.socket()
4 addr = socket.getaddrinfo('time-C.timefreq.bldrdoc.gov', 13)[0][-1]
5 print(addr)
6 s.connect(addr)
7 data = s.recv(100)
8 print(data)
9 s.close()

```

Note we are using port 13 to access this service.

The work on sockets should have convinced you that the ESP32 is just as able to access the Internet as a PC.

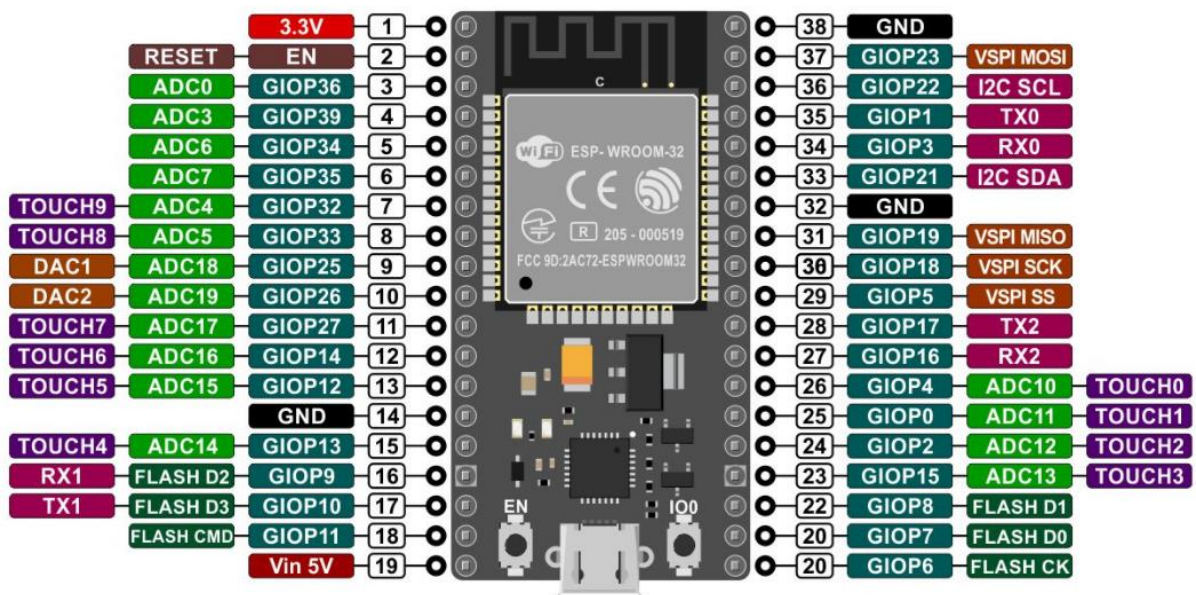
AS PROMISED IN CLASS, here is the LM35 code for you to try out. Take care with wiring.

```

1 # ESP32 LM35 code
2 from machine import ADC, Pin
3
4 import time
5 adc = ADC(Pin(36))
6
7 for i in range(20):
8     n=adc.read()
9     print(n)
10    V=n/4096
11    T=round(V*100, 1)
12    print(T)
13    time.sleep(1)

```







































ESP32 Pin Out



<https://forum.fritzing.org/t/esp32s-hiletgo-dev-board-with-pinout-template/5357>

The pins highlighted in green are OK to use. The ones highlighted in yellow are OK to use, but you need to pay attention because they may have unexpected behaviour mainly at boot. The pins highlighted in red are not recommended to use as inputs or outputs.

GPIO	Input	Output	Notes
0	pulled up	OK	outputs PWM signal at boot, connected to IO0 button, which can be used as user button after boot. 'Strap pin' which is read to determine boot mode
1	TX pin	OK	debug output at boot (this appears on Thonny on boot)
2	OK	OK	connected to on-board LED
3	OK	RX pin	HIGH at boot
4	OK	OK	
5	OK	OK	outputs PWM signal at boot
6	x	x	DO NOT USE connected to the integrated SPI flash
7	x	x	DO NOT USE connected to the integrated SPI flash
8	x	x	DO NOT USE connected to the integrated SPI flash
9	x	x	DO NOT USE connected to the integrated SPI flash
10	x	x	DO NOT USE connected to the integrated SPI flash

11			DO NOT USE connected to the integrated SPI flash
12			boot fail if pulled high
13			
14			outputs PWM signal at boot
15			outputs PWM signal at boot
16			
17			
18			DAC
19			DAC
21			
22			
23			
25			
26			
27			
32			
33			
34			input only
35			input only
36			input only
39			input only