Dyson School of Design Engineering, Imperial College London

DE1.3 Electronics 1

Lab 4B – ESP32, Sensors, Drivers

Peter Cheung, 15 June 2020

Introduction

Now that you have the ESP32 and MicroPython (uPy) environment set up, you can start programming the ESP32 in uPy to control electronic circuits.

It is important for you to appreciate the intended learning outcome for each Lab session, no more so than this Lab. I will be giving you four lectures on the applications of electronics in the next few weeks. They are entitled: Drive, Sense, Link and Source. In this Lab, you will experience the first two topics.

Furthermore, you will be using uPython to control things that are physical. Don't worry if you are not confident in Python programming and have forgotten your Autumn term Computing 1 module. In this lab, you will be learning through examples.

In this Lab, you will be using the Heltec wifi 32 kit module with Espressif's ESP32 microcontroller and a built-in OLED display. The board is a fully functioning computer with lots of different built-in peripherals (meaning it has all sort of electronics NOT part of the processor, but would help the processor to do useful things). A **microprocessor** designed to be embedded inside a piece of equipment, instead of going into a computer system, is called a **microcontroller**. The one here is based on an architecture known as **Xtensa XL6** by a US company **Tensilica**. This is different from the one we used last year with DE1, which was the more popular ARM process. ARM is the same processor used in almost all smart phones and tablets. However, due to remote working, I had to choose an alternative processor that is more suited for home laboratory.

For the remaining of Lab 4, you will be using the ESP32 with uPy only and you will work inside the PyCharm environment.

How to run MicroPython code on ESP32

There are two ways of running uPy code:

1) **REPL mode** – communicate with the EPS32 via the USB port and type uPy code directly after >>>
2) **uPy script stored in ESP32's flash memory** – the ESP32 run the uPy script directly from flash memory. This process is called "boot" (see later). In this method, you only need to apply power and press the reset button, the coding will run automatically.

PyCharm and MicroPython REPL mode

The easiest way to test uPy code is to simply type them in directly to the ESP32. This is done by:

1) Connect your ESP32 to your laptop using white USB cable I provided in your parcel. (Use this one because some phone charging USB – MicroUSB cable may have the

correct connectors at both ends, but they only supply power. The data wires may not be present.

2) Open PyCharm and open your project (which you previouly created in Lab 4A). This project should have been configured for MicroPython.
3) In the pulldown menu, click **`Tools > MicroPython > MicroPython REPL`**.
4) The bottom window will shown the REPL >>>. If not, press enter a few times and enter CTRL-C. (CTRL-C is an important key entry. It interrupts whatever program that the ESP32 is executing and return you to REPL mode).
5) Now you can type any valid uPy code and the ESP32 will execute it. For example, if you type: **`print('Hello world!'),`** this will appears in the bottom window.

Bootstrapping

The alternative is to run your uPy script without typing. Your code has to be stored in the flash memory inside the ESP32 chip. For this to work, you have to "flash" your program onto the chip beforehand.

When you first power up the ESP32 by connecting the USB cable to your laptop, the ESP32 go through a process known as **bootstrapping** – pulling itself up using the bootlace. What it does is to run the script "**`boot.py`**", then run "**`main.py".`**

I have set up the environment such that you will put in "**`main.py`**" the uPy instruction: **`execfile('`***`name_of_script`***`.py').`** In this way, you just edit this line in **`main.py`**, and you can execute any uPy script that you have written for different tasks. Otherwise, you will get confused by the many different versions of "**`main.py`**", each doing a different task. For example, you might create your code for Task 1 in the file task1a.py, and you must then also change main.py to contain: **`execfile('`*`task1a`*`.py').`**

Preparations

Take the following items from your stores:

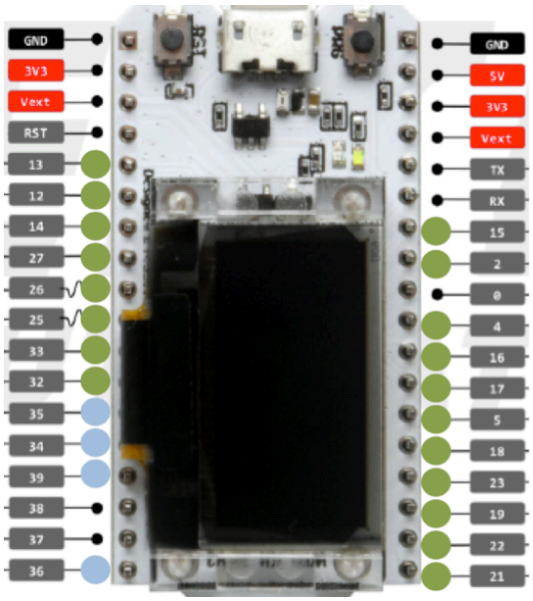
- 2x 100 ohm resistors
- Two light-emitting diodes (LED) in red and green
- push pin magnet
- 10k ohm potentiometer
- Rotary switch (already installed with SIG_GEN in Lab 0)
- MG90S servo motor
- DRV8833 motor driver (H-bridge)
- A DC motor with gear
- Neopixel LED strip (8-LEDs on a PCB)

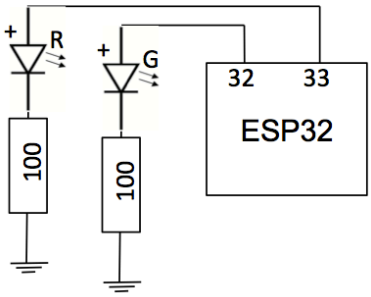
Task 1: Digital Outputs

In this task, you will learn to control ESP32 pins to output digital signals.

Most, but not all, of the pins on ESP32 can be programmed (or configured) to perform different functions. The diagram below shows which pins you can use for digital input or output (both purposes) – they are shown in GREEN. There are four pins which can ony be used as digital inputs (but not outputs). These are shown in BLUE.

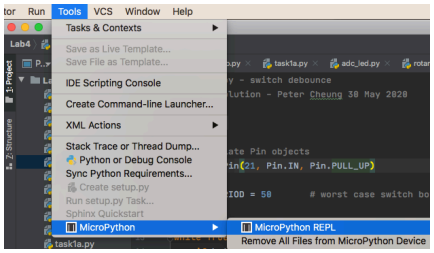
Step 1: Connect the RED and GREEN LEDs to the ESP32 as shown in the diagram.





Step 2: Start PyCharm and bring up >>> with:

`Tools > MicroPython > MicroPython REPL`.



Step 3: Type the following uPy code directly in response to REPL >>> . Line 1 is to import the Pin Class from the machine library. You can use: help("Pin") to investigate what the Pin Class contains. Lines 2 and 3 create two Pin objects, which are called rLED and gLED, both are digital output pins. uPy does not know or care what these two pins are used for.

```
1  from machine import Pin
2  rLED = Pin(33, Pin.OUT)
3  gLED = Pin(32, Pin.OUT)
```

Step 4: You can now turn the RED LED on and off using:

```
rLED.on()
rLED.off()
```

Step 5: You can even control the intensity of the LEDs by generating PWM signals on Pins 32 and 33 with the built-in class PWM.

Line 1: import Pin and PWM classes from uPy's machine library package.

Line 2&3: instantiate PWM objects on Pin 32 & 33, and set PWM frequency to 1000Hz.

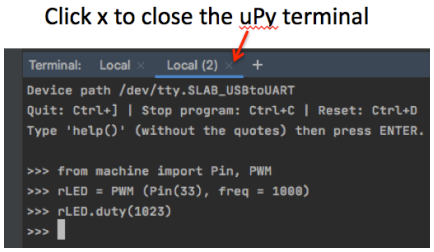
Note that the "duty" method is used to set the duty cycle of the PWM signal. However, the parameter used here is NOT in percentage. Instead the duty cycle is between 0 to 1023 (10-bit value). In other words, 100% duty cycle is specified by PWM.duty(1023).

```
1  from machine import Pin, PWM
2  rLED = PWM(Pin(33), freq = 1000)
3  gLED = PWM(Pin(32), freq = 1000)
```

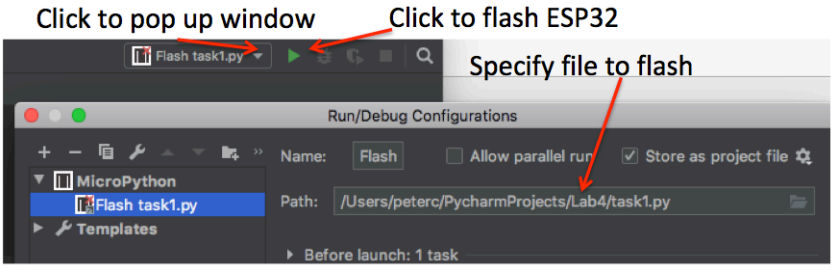
```
rLED.duty = (512)
rLED.duty = (0)
rLED.duty = (1023)
```

Step 6: Use the editor in PyCharm to create the file: "task1.py" which contain the following uPy script. (You can also download this from the course webpage, but it is helpful if you try to create this yourself with the editor.)

```
# task1: Digital output to LED
# rLED – pin 33, gLED – pin 32

from machine import Pin, PWM
import time

rLED = PWM(Pin(33), freq = 1000)
gLED = PWM(Pin(32), freq = 1000)

while True:       # Loop forever
    for i in range(0, 1023, 2):
        rLED.duty(i)
        gLED.duty(i)
        time.sleep_ms(20)
```



- Note that time.sleep_ms(20) will force the ESP32 to sleep for 20ms.
- Type CTRL+C to gain control of the uPy REPL again. Then close the uPy terminal, so that you can flash task1.py onto ESP32 over the USB cable.
- Edit configuration and specify task1.py as the file to flash.



- Modify the file "main.py" and change the file name to "task1.py". Flash "main.py" to ESP32. If you have problem flashing a new uPy script to the ESP32, please consult the Appendix at the end of this document.
- Open the uPy REPL terminal again.
- Enter CTRL-D to perform a soft reboot, and task1.py script will be executed. You will now see the red and green LEDs gradually lighting up from minimum to maximum intensity.
- Enter CTRL-C to interrupt the execute of task1.py. You will see the REPL again.
- You can control the intensity manually by entering uPy code such as:

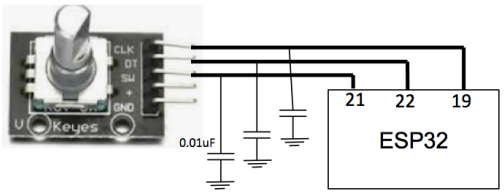
  rLED.duty(0)

  gLED.duty(1023)

Tips: uPy REPL >>> is really useful to debug programs and discover what is happening. For example, if you type: help(rLED), uPy will tell you that rLED is a PWM object, outputing the signal on Pin 33, with a frequency of 1000Hz and whatever duty cycle it is using at the time you interrupted the program. Further, it will show you all the method functions that are allowed with a PWM class object.

Task 2: Digital Input and switch debouncing

We are going to use uPy to read digital inputs provide by the rotary switch (which you used for SIG_GEN), which is connected to the ESP32 according to this diagram.



Step 1: While in uPy REPL terminal, try the following code.

```
1  from machine import Pin
2
3  button = Pin(21, Pin.IN, Pin.PULL_UP)
4  button.value()
```

Line 2 defines Pin 21 as an input pin. Pin.PULL_UP means that this pin has a pull-up resistor internally connected on the chip to the 3.3V power supply.

This pin is connected to the SW pin on the rotary switch. Therefore when the rotary switch is pressed, the is a short-circuit between SW pin and GND. The pull-up input mode of Pin 21 means that Line 4 will return a value of '1' (pull-up) when switch is not pressed, but '0' if the switch is pressed.

Step 2: Write your own uPy script task2a.py which uses the switch to turn on and off the rLED – SW pressed ON, SW not pressed OFF. Flash this to the EPS32 and verify that it works.

Step 3: Type the script task2b.py shown on the right. Flash this to the ESP32. Remember, you have to enter CTRL-C to break the while True: loop in the ESP32, gain back control (i.e. see the REPL >>>), close the uPy terminal before you can flash another program.

```
1  # task2b.py – switch bouncs effect
2  # Peter Cheung 30 May 2020
3
4  from machine import Pin
5
6  # instantiate Pin objects
7  button = Pin(21, Pin.IN, Pin.PULL_UP)
8
9  while True:       # Loop forever
10     if button.value() == 1:
11         print('+', end = '')
12         while button.value() == 1:
13             pass
14         print('-', end = '')
```

This program will detect SW being pressed or released, and output a '+' or '-' character on the uPy REPL terminal. Now remove the 0.01uF capacitor installed on Pin 21 to ground. You should now see the **effect contact bounce** of the switch. The 0.01uF performs the function of debouncing. Discuss this with your Team members about contact bounce.

Step 4: Modify task2b.py to task2b.py, which performs debouncing in software (instead of using a capacitor). You can do this by detecting a change in Pin 21 value, wait for a certain period for the contact bounce to die down before reading the Pin 21 value again.

Task 3: Rotary Switch for control

You will now learn to use the rotational function of the rotary switch to control intensity of the LED.

Step 1: Download the two rotary switch driver files from the course webpage (rotary.py and rotary_irq_esp.py). Flash them to the ESP32.

Step 2: Enter or download the following program: task3a.py.

```
1  # task3a.py
2
3  import time
4  from machine import Pin
5  from oled import OLED  # load oled module from flash memory
6  from rotary_irq_esp import RotaryIRQ
7
8  oled = OLED()  # Instantiate an OLED object
9  oled.init_display()
10
11 r = RotaryIRQ(pin_num_clk=19,
12                pin_num_dt=22,
13                min_val=-100,
14                max_val=100,
15                reverse=True,
16                range_mode=RotaryIRQ.RANGE_BOUNDED)
17
18 val_old = r.value()
19 while True:
20     val_new = r.value()
21
22     if val_old != val_new:              # knob turned
23         val_old = val_new
24         oled.draw_text(32, 32, '{:4d}'.format(val_new), size=2, space=2)
25         oled.display()  # flush display
26         time.sleep_ms(20)
```

Line 11 – 16 instantiate the rotary switch object. The first four parameters are obvious. "reverse" determine the direction of rotation for increase or decrease values. "range_mode" is set to RANGE_BOUNDED, which is bounded by min_val and max_val. The other two settings are RANGE_UNBOUNDED and RANGE_WRAP.

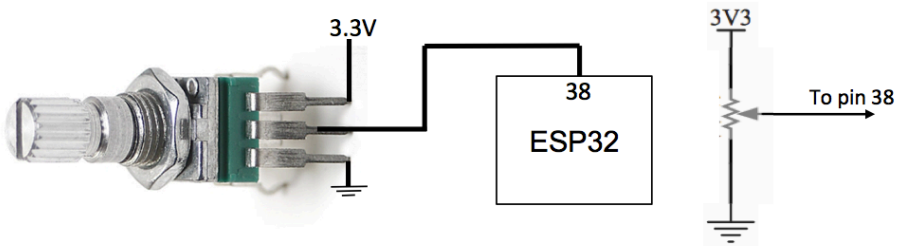
Line 24 shows how to use formatted text where {:4d} specifies a 4 digital decimal integer.

Step 3: Flash and test task3a.py.

Step 4: Modify task3a.py to task3b.py so that you use the rotary switch to control the intensity of the LEDs. Flash and test task3b.py.

Task 4: Analogue Input and the Analogue to Digital Converter

In this task, you will use the potentiometer to produce a voltage between 0 and 3.3V, and learn to use the ADC in the ESP32 to convert this voltage into a digital number.



Step 1: Connect the 10kW potentiometer as shown to the ESP32.

Step 2: Enter the code above using the uPy REPL interactively. Use the up-arrow to recall the command in Line 9 repeatedly to read the potentiometer voltage while turning the spindle of the potentiometer.

```
3  from machine import Pin, ADC
4
5  pot = ADC(Pin(38))
6  pot.atten(ADC.ATTN_11DB)
7  pot.width(ADC.WIDTH_9BIT)
8
9  pot.read()
```

Line 5 configures Pin 38 to be an analogue input pin connected to ESP32's internal ADC. You may use Pin 32-38 as ADC pin.

Line 6 allows input voltage to be in the range of 0.0V – 3.6V. Without attenuation by 11dB, the input range is limited to 0.0V to 1.0V.

Line 7 specifies that the ADC converted voltage to be 9-bit in width. Therefore the converted voltage value will fall within the range of 0 to 511.

Step 3: Another source of analogue input signal is through the Hall Effect sensor, which is built into the ESP32. This sensor detects the presence of a magnetic field and provide an analogue signal that is converted into a digital number that can be read by the user.

Enter via the uPy REPL interactively the following code segment:

```
import esp32, time

while True:
    print(esp32.hall_sensor())
    time.sleep_ms(500)
```

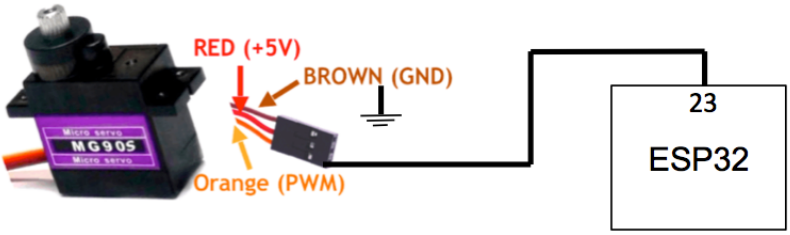
Note that you can use the backspace key to reduce the identation level. You should see in the uPy terminal window the Hall Effect Sensor reading. Now place the push pin magnet on top of the OLED display. You should see the sensor reading increase from the previous value. Unfortunately the push pin magnet I provided in your parcel is not that strong. Therefore the difference in the reading may not be that large. Furthermore, the Hall Effect Sensor built into the ESP32 chip is not very accurate. Therefore the reading will fluctuate.

Step 4: Write your own task4.py script to use the push pin magnet as a switch to turn the red LED on and off.

Task 5: Servo Motor

To drive a servo motor, you should use a 50Hz PWM signal. Remember that the PWM duty cycle parameter is from 0 to 1023.

Step 1: Connect the servo motor to the ESP32 as shown here. Note that you will have to use the 5V power supply, not the 3.3V. Install the pointer arm on the motor axis.



Step 2: In uPy REPL mode, enter the following code interactively. The servo arm should be at the motor position.

```
from machine import Pin, PWM

servo = PWM(Pin(23), freq = 50)
servo.duty(72)  # 0 degree
```

Step 3: Interactive explore the value of the duty cycle to find the two extreme angle of the servo motor. This value should be roughly in the range of 20 to 124, corresponding to -90 and 90 degrees.

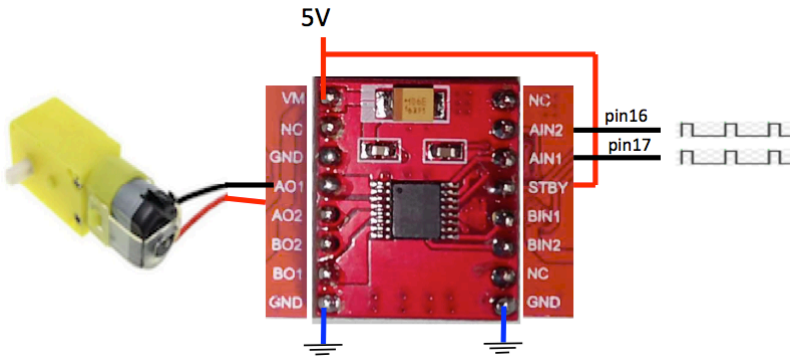
Step 4: Write the uPy script task5.py that uses the rotary switch to control the angle of the motor from -90 to 90 degree. Flash your program to the ESP32 and test it.

Task 6: DC motor and the H-bridge motor driver

In this task, you will drive the DC motor using the DRV8833 H-bridge module provided. You need the H-bridge module because the ESP32 output does have enough current drive capability to drive a DC motor. Furthermore, the DC motor need a 5V supply and the ESP32 output is only 3.3V.

Similar to LED, motor speed is controlled using a PWM signal. The H-bridge module has two motor control signals: AIN1 and AIN2. These two input pin should be connected to ESP pin 17 and pin 16 respectively. The H-bridge output pins are AO1 and AO2. These should be connected to the two wires of the DC motor.

Step 1: Construct the circuit shown here.



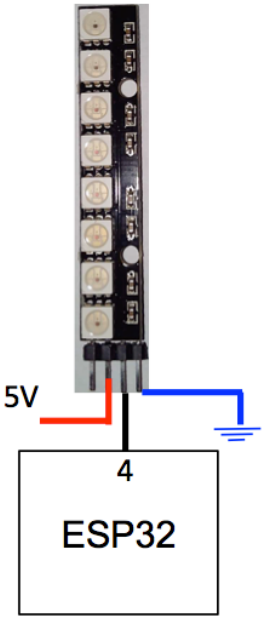

```
from machine import Pin, PWM

A1 = PWM(Pin(17), freq = 1000)
A2 = PWM(Pin(16), freq = 1000)

A1.duty(1023)    # forward
A2.duty(0)

A1.duty(0)       # reverse
A2.duty(1023)
```

Step 2: Enter the above code in the uPy REPL terminal. You should be able to control the speed and direction of the motor by varying the duty cycle parameter of A1 or A2 PWM signals between the range of 0 to 1023.

Task 7: Neopixel strip

In this final task, you will learn how to control the neopixel LED strip in uPy.

Connect the following circuit. Make sure that you connect the 5V and GND to the correct pins. (Double check your connection before you connect the USB cable.)



Enter the following lines of code in the uPy REPL terminal. This will light up the bottom three LEDs with different colours and intensity.

```
from machine import Pin
from neopixel import NeoPixel

np = NeoPixel(Pin(4), 8) # 8 pixels in strip, Data on Pin 4
np[0] = (255, 0, 0)      # Bottom pixel is (R, G, B) red intensity 100 max 255
np[1] = (0, 128, 0)      # Green at 1/2 intensity
np[2] = (0, 0, 64)       # Blue at 1/4 intensity
np.write()
```

As can be seen here, controlling neopixels are very easy in uPy. All you need to do is to set up the array one element at a time with a tuple of values between 0 and 255, with each tuple corresponding to R, G and B colours (i.e. (R,G,B)).

For details, see the Section 11 of the MicroPython document:

https://docs.micropython.org/en/latest/esp8266/tutorial/neopixel.html

Appendix – Flashing a .py file to ESP32 – Tips, hints and work-around

Your scripts (e.g. boot.py, main.py, task1.py etc) need to be programmed to the flash memory on the ESP32 module. To do that you need to run the esptool.py and other utilities to erase and program the flash memory. Doing it from PyCharm is a possible and convenient way. However, we discover that for some unknown reason, you may encounter difficulties.

When the ESP32 was programmed to run in an infinite loop, on some laptops, flashing a new script from PyCharm fails to take over control of the USB port to perform the flashing. This appears more common on PCs than in Mac (although a few Macbook users also experience the same problem). Even if they used CTRL+C to force ESP32 in REPL mode, they still have this error message:

```
Connecting to /dev/tty.SLAB_USBtoUART
Uploading files: 0% (0/1)
/Users/vxu/PycharmProjects/HomeLab4/oled.py -> oled.py
b'r'
Traceback (most recent call last):
  File "/Users/vxu/Library/Application Support/JetBrains/PyCharm2020.1/plugins/intellij-micropython/scripts/microupload.py", line 138, in <module>
    main(sys.argv[1:])
  File "/Users/vxu/Library/Application Support/JetBrains/PyCharm2020.1/plugins/intellij-micropython/scripts/microupload.py", line 79, in main
    files.put(remote_path, fd.read())
  File "/Users/vxu/PycharmProjects/HomeLab4/venv/lib/python3.8/site-packages/ampy/files.py", line 208, in put
    self._pyboard.enter_raw_repl()
  File "/Users/vxu/PycharmProjects/HomeLab4/venv/lib/python3.8/site-packages/ampy/pyboard.py", line 192, in enter_raw_repl
    raise PyboardError('could not enter raw repl')
ampy.pyboard.PyboardError: could not enter raw repl
```

The reason for this appears to be the ESP32 is still busy running the infinite while loop from the previous flashed programme. It will NOT yield control to allow flashing to be performed. The solution is to delete the main.py file so that ESP32 has nothing to do and will respond to PyCharm when asked to flash a new script.

To remove main.py, go to Tools > MicroPython > MicroPython REPL to bring up the REPL terminal. Enter

>>> import os

>>> os.remove(“main.py”)

Now close the REPL window. Flash you .py scripts and flash main.py last. This should work.