The Micropython Microcontroller

Aims of this experiment

- Explore the capabilities of a modern microcontroller and some peripheral devices.
- Understand how the signals at the various interfaces work with the software.
- Make a simple measuring instrument using a microcontroller.

The early parts of this experiment are fairly prescriptive, to get you started. The later parts (3.3 onwards) provide scope for you to devise your own microcontroller application. There is no need to do all the later parts in equal detail. Quality beats quantity.

Introduction

Microcontrollers are essentially single-chip computers. They are particularly useful for processing electrical signals, with which they can connect quite simply. They can simplify electronic circuits in many applications, because the hardware is fixed, and a new application requires only software changes.

A very small microcontroller, costing ∼30p, might have the processing capacity to control a complex multi-way light switch. Capabilities and prices range upwards by perhaps 100 fold; beyond that lie smartphones and PCs. But the unique feature of microcontrollers is that they are intended for use in embedded systems, in which they behave as a sophisticated electronic component, rather than providing general-purpose computing.

The microcontroller in this experiment is an "STM32" series device, which is based on the ubiquitous 32-bit ARM processor. It has 120 kilobytes (K) of working memory, 512 K of permanent (flash) memory, and runs at 96 MHz. By microcontroller standards, these are large numbers. The advantage of this level of device is that it can run a specialised version of the high-level language Python in real time. This is a great convenience for experimental development.

The exercises in the experiment introduce some of the ways the microcontroller can be connected to other electronics, and the kind of devices you can build by programming. This should be useful to you in future experimental work (e.g. Year 2 projects, group studies...)

1 The MuNu ($\mu\nu$) development environment

Plug the Micropython board ("PyBoard") into a USB socket on the PC, and start the MuNu Micropython Integrated Development Environment (IDE). Click the Repl[1] button to open a command line connection to the board. You should see the Micropython prompt, `>>>` from the board. Test it by typing something like `print(2*3)`. You could type a whole program here, but it is more convenient to edit and save it in the editor window.

Open a new file in the editor, and enter in it a simple Python program, for example

[1] REPL = Read, evaluate, print loop. i.e. the software behind the command line

```
for i in range(10):
    print('{} * 5 = {}'.format(i, i*5))
```

Use the RUN button to run your program. It runs *on the microcontroller board*, while the PC just serves as a communications terminal. Try saving and loading your program. Note that the editor saves and loads files *on the PC*. It is useful to know that on the Repl command line, Ctrl-C interrupts the running code and Ctrl-D restarts the Python interpreter.

2 Controlling an output

As with any embedded project, the first thing to get working is turning on and off a light-emitting diode (LED). Connect an LED and a 270 Ω series resistor between pin **`Y1`** and ground (**`GND`**), using the breadboard, as shown in Figure 1. The short LED lead is the cathode, and should connect to ground. Also connect an oscilloscope so you can see the signal coming out of the microcontroller.

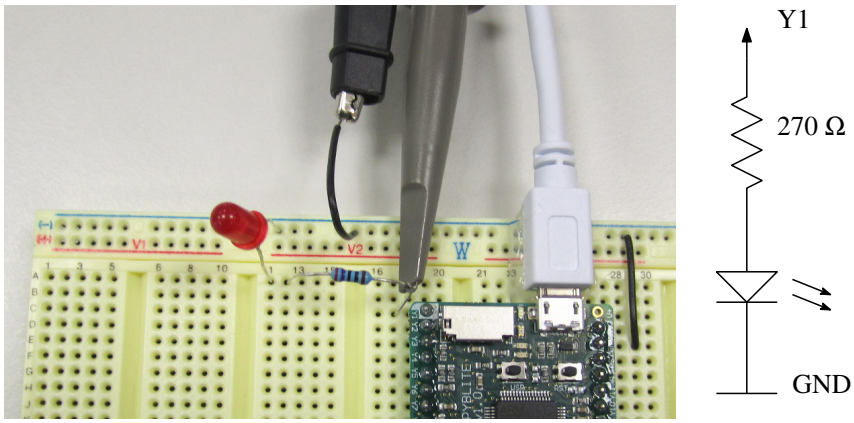

Figure 1: LED and oscilloscope probe connected to Y1

2.1 Flashing LED

Open the program **`flash1.py`**, and run it using the RUN button. The LED should light up. If not, check your wiring. Set up the oscilloscope to show the pulsed waveform being produced.

flash1.py

```
from pyb import Pin, udelay

myled = Pin('Y1', Pin.OUT_PP)

while True:
    myled.value(1)
    udelay(100)
    myled.value(0)
    udelay(100)
```

All our programs will import from the **pyb** module, because it gives access to the hardware features of the PyBoard (which contains the microcontroller). **Pin** is a class that controls the breadboard pins; the instance we keep in **myled** connects to pin **Y1**, and configures it as a digital output in "push-pull" mode[2] The **udelay** function introduces a delay specified in microseconds.

You should be able to work out how the code gives rise to the signal seen on the oscilloscope. Check your understanding quantitatively by altering the delays. What happens if you leave the delays out completely? **Q**

What does this tell you about the execution speed of Python statements? Record your observations and some illustrative waveforms.

Why does the LED appear to be constantly on, while the signal is clearly switching on and off? How slow must the pulses be before you can see the LED flashing? **Q**

2.2 Faster flashing

The program **flash2.py** has been written to run a bit faster, by using local variables, and looking up the **value()** and **udelay()** methods in advance of the loop. What is the speed improvement? **Q**

2.3 Much faster flashing

The program **flash3.py** has been written with the main loop in ARM machine language rather than Python. This is considerably harder to write, and there is no need for you to understand it, but it does indicate the ultimate speed of the microcontroller. How fast does this go, and how much faster does a single machine instruction run than a Python statement? You may need to set the oscilloscope probe to x10 (and compensate with the sensitivity control) to see such a fast signal properly. **Q**

2.4 Why the resistor?

Q

It is important to include a resistor between a driving voltage and any LED. Use the extract from the LED data sheet in Fig. 2 along with Ohm's law to work out what would happen if you connected an LED to 3.3 V without a resistor. What current would you expect when using the 270 Ω resistor in this experiment?

[2]The pin voltage depends on the value set: value == 1 ⇒ 3.3 V (pushing current out); value == 0 ⇒ 0 V (pull)

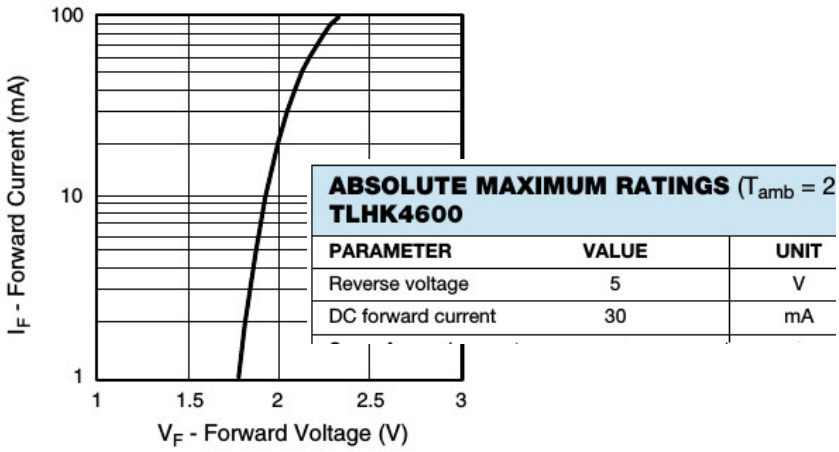

Figure 2: Red LED characteristics (extract)

3 A digital display and digital input

3.1 Seven-segment display

To get useful information out of the microcontroller, connect the four-digit, 7-segment display to the 3.3 V power, ground, and pins X9 and X10, as shown in Fig. 3. Take care to get the pins connected in the right order!

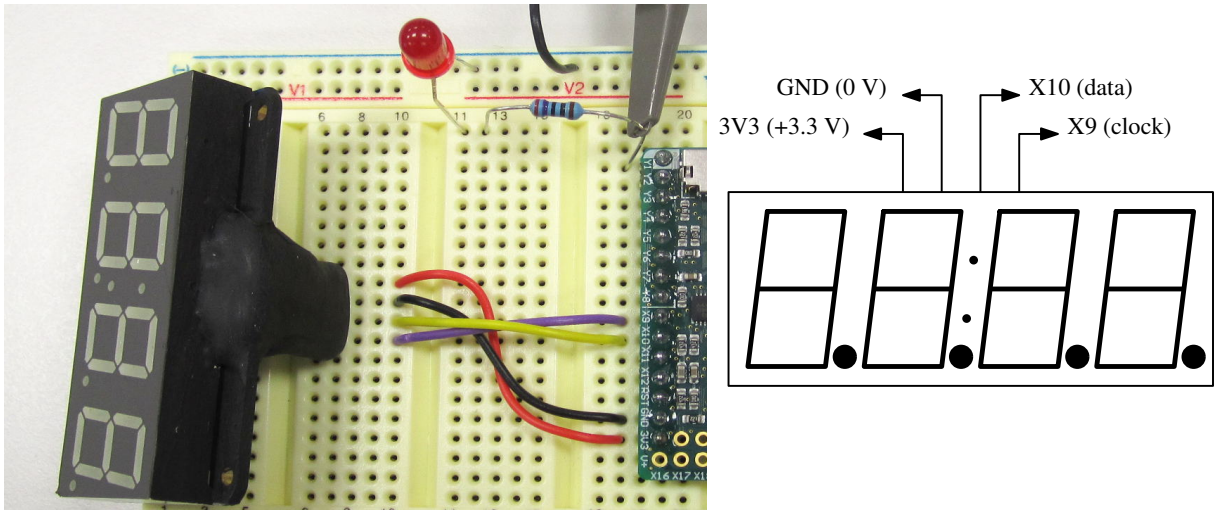

Figure 3: Seven-segment display connections

Use the program **`count.py`** to test the display. The operation of the **`while`** loop should be obvious. You might wonder how the **`display.show()`** method works, given that is has to control around 30 LEDs (all the display segments and dots), using only two wires, **`X9`** and **`X10`**. The advantage of putting

all the code for this in a *class* [3] : **`Sevensegx4`** is that you (as, say, the author of **`count.py`**) don't have to know. Nevertheless, you should find out enough about the "I^2C bus" which is implemented on **`X9`** and **`X10`** to write short explanations of:

- How the clock and data wires enable a stream of data bits to be sent. **Q**
- Roughly how much time it takes to send each bit.
- How the display knows the bits are intended for it, rather than for any other device on the bus.

You may find it useful to look at the the clock and data waveforms with the oscilloscope.

3.2 Input from a switch

Leaving the display connected, now add a switch to the circuit, between pin **`Y11`** and ground, as shown in Fig. 4. Run the program **`countswitch.py`**. Try the action of the switch.

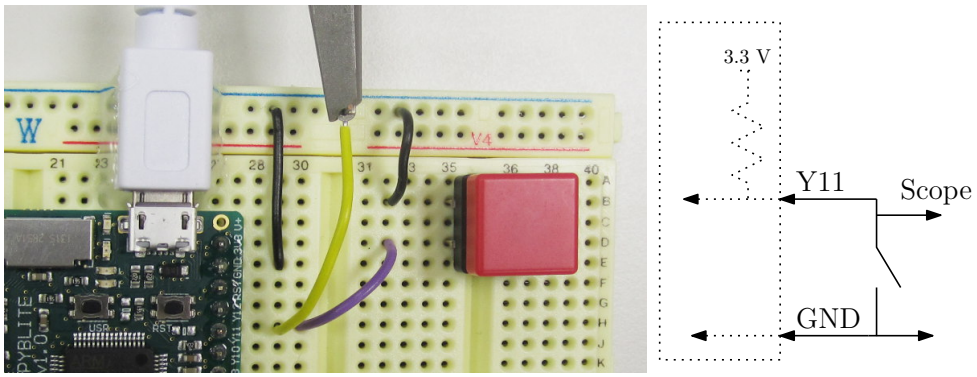

Figure 4: Wiring of the switch. The dotted components are inside the microcontroller

`countswitch.py` contains the line **`switch = Pin('Y11', Pin.IN, Pin.PULL_UP)`**. This makes an instance of the Pin class. The initialisation specifies the pin number, sets the pin as an input, and connects an internal "pull-up" resistor as shown in the figure. How it does all this is, again, hidden inside the class definition. Explain what logic levels you expect at pin **`Y11`** when the switch is open and closed. Check with the oscilloscope. **Q**

The part of the code that deals with the switch is

switch code

```
while switch.value() == 0:
   pass
```

Explain how this works to freeze the count when the switch is pressed. **Q**

[3] Briefly, a class is a collection of variables and functions that work together to accomplish some task. All the user of a class needs to know is what its externally visible functions ("methods") accomplish; only the author of the class needs to see the details.

3.3 Reaction timer

Using the same circuit as in the previous section, run **`reaction1.py`** It is a simple reaction timer that makes the user wait for a short random time, then starts counting. Pressing the switch stops the count and thus displays a number proportional to the user's reaction time.

Look at the code in **`reaction1.py`** and answer the following questions, in terms of the code used: **Q**

- How long are the hyphens displayed for?
- What does the break statement do here?
- What happens if the switch is never pressed?

The unit of time in **`reaction1`** is however long one cycle of the **`for count...`** loop takes. Use an external stopwatch to calibrate it, and alter the code so that it reads out in milliseconds (approximately). Document how you did it. What is your reaction time?

You might notice that you can cheat and get a reaction time of zero. If things are going well, consider how this could be fixed.

3.4 Binary coding - displaying text

The **`Sevensegx4`** class knows how to display numbers, but not arbitrary symbols. It does however have a method **`showraw()`** that lets you turn on any combination of display segments. Its argument is a list of one to four numbers, where each bit in the binary representation of each number controls one display segment, as shown in Fig. 5.

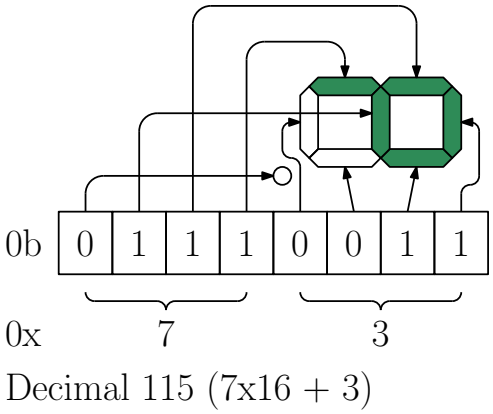

Figure 5: Display encoding of the letter P, either as binary 0b01110011, hex 0x73 or decimal 115.

You can try this at the command line, as follows. If you type a Ctrl-C, while running **`reaction1.py`**, then the program will stop, but the variables like **`display`** will persist, so you can try some live experiments, e.g.

```
display.showraw([0b01110011, 0x30, 121])
```

Python lets you express integers in binary (0b...), hexadecimal (0x...) or decimal (digits 0-9), as convenient. You would not normally use all three systems in the same line, except when writing an example.

Work out how to display the message **Good** on the display. If things are going well, make **reaction1.py** display "Good" (or perhaps "bAd") depending on the measured reaction time.

4 The Neopixel display

"Neopixel" is a trade name for a multicolour LED with a built-in controller chip that allows many such LEDs to be connected in a daisy chain, all controlled by a single signal that passes from each LED device to the next in line.[4] Unlike the seven-segment display, there just a data line, and no clock. Thus the timing of the bit sequence on the data line is defined by the manufacturer, and all users must adhere to it. Again, this has been done for you in a Python class. Connect up the ring of neopixel devices, as shown in Fig. 6,

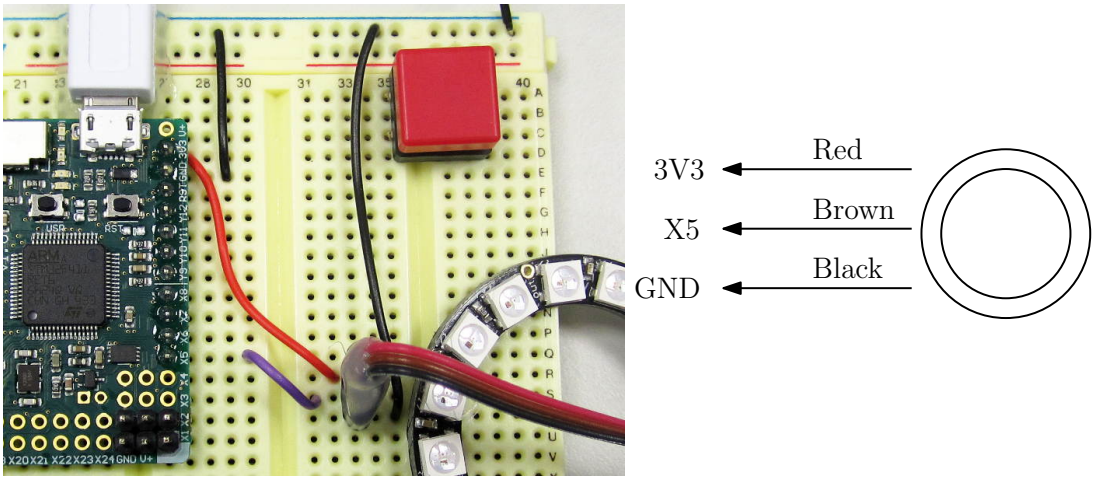

Figure 6: Connections to the Neopixel ring

and try it out using the program **chase1.py**.

Connect the oscilloscope to the **X5** data line to examine the data bits being sent while **chase1.py** is running. Each bit occupies a fixed time slot, with zeros being sent as short pulses, and ones as long pulses. How long is: **Q**

- a zero?
- a one?
- the interval between bits?

As with the seven-segment display, the appearance of a single element is determined by a numerical code. The lowest 8 bits of the number control the brightness b of the blue LED, the next 8 bits the red, r, and the top 8 bits green, g. Thus, if r, g and b are each in the range 0-255 (the maximum for 8 bits) the number is `(256 * 256 * g) + (256 * r) + b = (g << 16) + (r << 8) + b` since multiplication by 256 is equivalent to shifting 8 bits to the left. [5]

Ctrl-C out of **chase1.py** and try the following to make LED number 8 red:

[4] The original manufacturer's name is WS2812

[5] Actually, the Neopixel class ignores the top bit in order to limit the current, so the practical limits are 127, not 255

Manually changing the neopixels

```
ring[8] = 80 * 256
ring.update()
```

Now make it yellow. How about making all the LEDs yellow? Document your changes.

5 Analogue inputs and measuring instruments

5.1 Accelerometer

The board contains a 3-axis accelerometer, made available by the **`pyb.Accel`** class. Again making use of the seven-segment display, run the simple example **`accel1.py`** to display readings from the accelerometer. The **`filtered_xyz()`** method provides three orthogonal components of acceleration, which the code calls **`ax`**, **`ay`** and **`az`**. As you tilt the board, you should see the effect of gravity on the accelerometer. Modify the code to make your own accelerometer instrument. You could, for example make it display in units of the Earth's gravitational acceleration, g, or make an angle measuring instrument, see Fig. 7. The Python arctangent function, **`math.atan2()`** will be helpful for this.

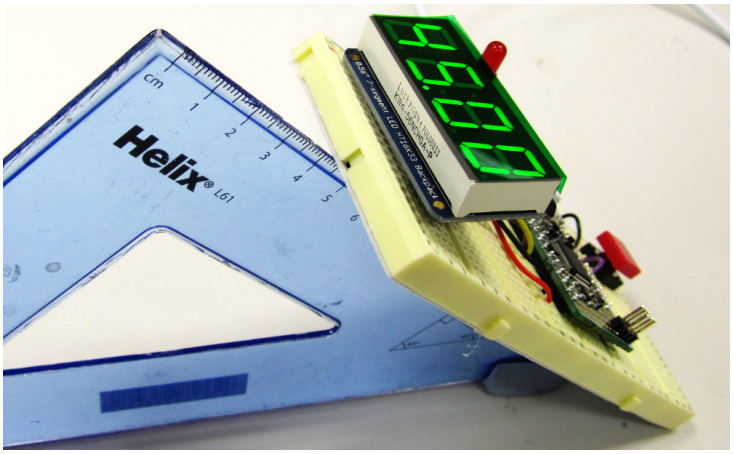

Figure 7: A tilt meter in use

5.2 Light meter

The microcontroller has an on-chip analogue-to-digital converter for digitising the signal from external sensors. Connect a potential divider made using a light-dependent resistor (LDR), as shown in Fig. 8

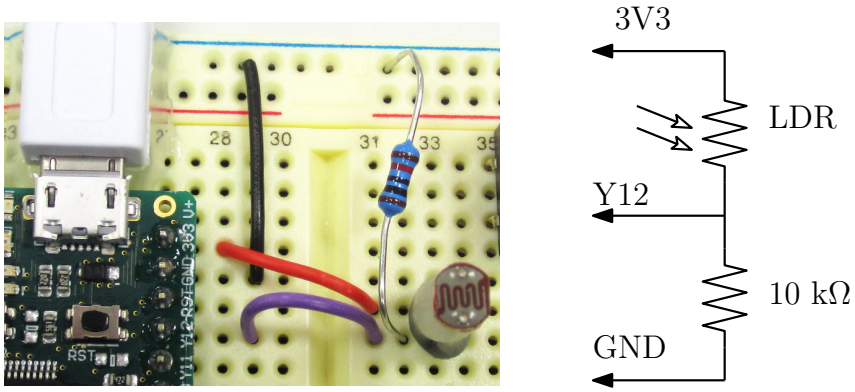

Figure 8: Connections to the light-dependent resistor

The code in **`light1.py`** uses the **`ADC`** class to get a reading of the voltage at pin Y12, which it then displays on the seven-segment display. Modify the code to produce an analogue readout on the Neopixel ring. Options include a speedometer-type display or a variable-speed chasing light.

6 Checklist

Your lab book should contain:

- Diagrams of circuits you built
- Programs (or parts of programs) that you wrote or understood, with comments showing how they work – but there is no need for the full text of programs that are supplied to you.
- Observations and explanations as prompted by this manual.
- Enough detail to enable someone to quickly pick up the thread of what you did and what you found out.

MSC 2017-09-23