## Introduction

In the last lab you did your first activities that involved both input from the environment (a button) and output (LED's and a buzzer.) That was just the beginning! You are going to learn about and build many more devices[†] that do input and output. The outputs and sensors on the PicoW are very limited. It can only measure its own temperature and blink an LED. The real power and flexibility of a microcontroller shows when when you extend its capability by adding external controller and sensors. There are hundreds of sensors and controllers easily available, so what you can do with a microcontroller device is only limited by your imagination.

[†] By *device* I usually mean a standalone microcontroller that does some sensing, controlling, and communication. For example this week you are going to build a temperature data logger. Once it is finished, you only have to power it up and it will record the temperature and store the data on the microcontroller. It does not have to plugged into a computer.

## Chapter 7 – Burglar Alarm

Carefully read Chapter 7 and build a burglar alarm. Note that this an *infrared sensor*. It will work in total darkness.

- **Challenge**: Pair up with someone in the class and add a second alarm and print which one is being triggered.

## Chapter 8 – Temperature Gauge

This chapter introduces an important topic: Analog-to-Digital Conversion (ADC.) ADC is when a continuous analog voltage is read by a microcontroller. Up until now every thing was digital – either **ON** or **OFF** . In this chapter you will be reading a temperature sensor which can have a wide range of values.

You will also be doing some Digital-to-Analog Conversion (DAC.) Instead of turning an LED either **ON** or **OFF**, you will fade it to dimmer values.

## *Notes on ADC and More*

Sometimes in the lab, I will insert a mini-lecture on a topic. The first is on bits, bytes, and hexadecimal.

### *Bits, Bytes, and Hexadecimal*

Fundamentally a microcontroller can only read and write 0's and 1's. It can string them together into a *binary* number. The binary number is (almost) like what you use all of the time in decimal, or base 10 numbers. Figure 1 shows a decimal integer. Each digit can have a value from 0 to 9, and each column is a power of ten greater: $10^0$, $10^1$, $10^2$, $10^3$, $10^4$, in this example.

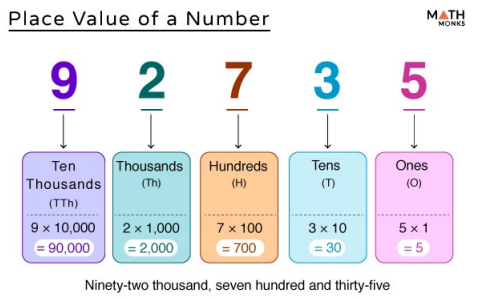


*Figure 1: Decimal number system. Digits go from 9 to 0. Successive decimal places are increasing powers of 10.*

Binary numbers work in the same way but each digit can only be 0 or 1, and the place are powers of 2, $2^0 = 1$, $2^1 = 2$, $2^2 = 4$, etc. Figure 2 show how the number 202 is represented in binary. Notice it takes an 8 digit number to represent 202, a 3 digit number in decimal!

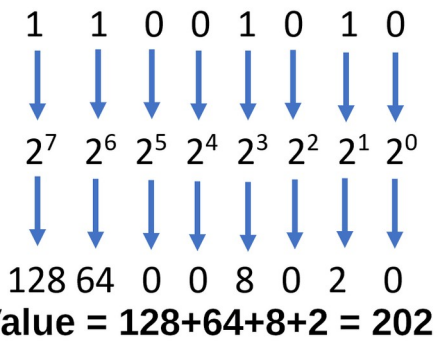


*Figure 2: Binary numbers. The value are either 0 or 1, and the places are increasing powers of 2.*

In the textbook it tells you that the ADC on the PicoW is a 12-bit ADC. That means it outputs 12 binary digits so the output can be from 0 to $2^{12}$-1, or 0 to 4095. A peculiarity of uPy is that all ADC are *shifted* to go from 0 to $2^{16}$-1, or 0 to 65535. For example if the ADC outputs the binary number **`0b010101010101`** (binary numbers start with a '0b' or '0B') get padded at the right to be **`0b0101010101010000`**, so instead of reading 1365, it reads 21,840.

OK. It is hard to remember numbers like 4095, and 65535, so experiences programmer use a third system, *hexadecimal*. You might guess hexa=6 and decimal=10. This is a base 16 number system, and the one I will use in my notes. Hexadecimal digits from from 0 to f and successive places are powers of 16. Since it is a large base than decimal, it takes fewer digits to express a large number.

The most compelling reason is the easy translation from binary. Almost all binary features of microcontrollers are in groups of 4 binary digits. For example, computers used to all be 32-bit processors, newer ones are 64-bit. Later in the course we will use 16-bit DAC's. In programming, languages use 8-bits for characters, 16-bit for small integers and 32- or 64-bits for large integers. Table 1 shows the correspondence between binary numbers and hexadecimal digits.

| | | | |
|---|---|---|---|
| b0000 | 0x0 | b1000 | 0x8 |
| b0001 | 0x1 | b1001 | 0x9 |
| b0010 | 0x2 | b1010 | 0xa |
| b0011 | 0x3 | b1011 | 0xb |
| b0100 | 0x4 | b1100 | 0xc |
| b0101 | 0x5 | b1101 | 0xd |
| b0110 | 0x6 | b1110 | oxe |
| b0111 | 0x7 | b1111 | 0xf |

*Table 1 – Binary number to hexadecimal digits.*

If you look at the binary number example, and the table you can see the corresponding hexadecimal number is 0xca. This may seem awkward at first, but it makes some things easier. For example, the 12-bit ADC runs from 0 to 0xfff. Why? 12 bits is a 3 digit hex number, and the maximum hex digit is f, so you get 0xfff.

The book uses the decimal equivalents, but I expect you to translate those weird decimal numbers to hex in your code.

**Hint**: python can convert for you. **`hex(255)`** returns **`0xff`**. There are also functions **`int()`** for decimal, and **`bin()`** for binary.

**Exercises**:

- What is the maximum value of an 8-bit ADC in decimal, binary, and hex?

At a low level, numbers are sometimes *shifted* right or left by some numbers of bit. If shifted left, 0's are *pushed* in from the right. For example shifting **`0b1011`** once to the right results in **`0b10110`**. In python you can test this by typing **`bin(0b1011 << 1)`** at a python prompt. Go ahead!

When a number is shifted right, the digit just disappear, so **`0b101101 >> 2`** is **`0b1101`**.

- ▸ What are the results of **`0xff<<4`** in binary, decimal, and hex?
- ▸ What are the results of **`100 >> 2`** in binary, decimal, and hex?
- ▸ What are the results of **`100 << 1`** in binary, decimal, and hex?

- It seems like shifting an integer left multiplies it by two. Explain this by writing 10 and 20 in binary and describing what is happening.

## *ADC's and resolution*

The number if bits used in an ADC or DAC is called its resolution. In code you will usually convert the input from and ADC to volts. The formula used is

$$V_{adc} = V_0 (value / ADC_{max})$$

, where $V_0$ is the maximum voltage the ADC can convert, *value* is the integer read by the ADC, and $ADC_{max}$ is the maximum value the ADC can output. The microcontroller data sheet will tell you both $V_0$ and $ADC_{max}$. On some microcontrollers you set both of these values. For the PicoW, $V_0$ = 3.3 V, and $ADC_{max}$ = 0xfff (12-bits.) (See how easy it is to figure that in hex?)

The second important parameter of the ADC is its *resolution*. The resolution is the small step in voltage that will increment the ADC output. The resolution is Resolution = ( Operating voltage of ADC ) / 2^(number of bits ADC)

$$\Delta V = V_0 / ADC_{max}$$

For the PicoW this is 0.81 mV. This means if you put in a continuously increasing voltage to the ADC, the your measurements will look like a staircase of 0.1 mV steps. This is important because in science you always want to know how accurate a measurement. So below we will use this to calculate the accuracy of a temperature measurement.

## CHAPTER 8 CHALLENGES

- When you code the potentiometer, use hex instead of decimal to convert the ADC to a voltage.
- **Challenge**: Use an DMM (Digital MultiMeter) to measure the voltage at the wiper of the pot (short for potentiometer.) Make a table for different pot positions of the ADC output (in hex), the converted ADC voltage, and the DMM reading. How close on average is the ADC to the DMM? Include the minimum and maximum settings of the pot.
- **Challenge**: Is your pot linear or log? You can guess by taking readings at about 0, ¼, ½, ¾, and 1 positions in the range of the pot movement.
- **Challenge**: Code the program that measures the temperature of the PicoW (pg. 98ff.) Add the resolution and units to your print out. Your output should look something like `41.0 +/- 0.5 C`. The general rule of reporting numerical results in the lab should be: 1) Round the uncertainty (resolution) to 1 significant figure, then 2) round the measurement to the same decimal place.
- **Challenge**: You instructor will give you a TMP36 temperature sensor. It is a small black semicircular case with three wires. When you look at the flat face of the sensor, the wires from left to right are Vdc, Aout, anf GND.

  ***Be careful not to mix Vdc and GND or you will release the genie smoke!***

  ***(and genies can't be put back in the bottle!)***

  - Wire the output voltage to one of the ADC pins on your PicoW.
  - Get the formula to convert from temperature to Celsius from the TMP36 data sheet. Use that in your program
  - Modify the CPU temperature program to print the temperature of the sensor. You can easily test it by holding the sensor between your finger and thumb.
  - **Challenge**: Make an led get brighter or cooler when the sensor temperature goes up and down.

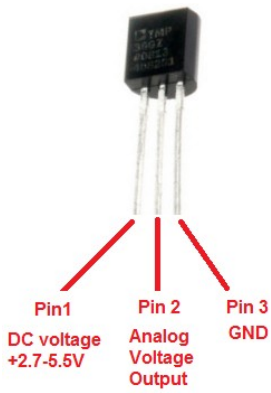


*Figure 3: The TEMP36 sensor.*

## CHAPTER 9 – A DATA LOGGER

In this chapter you will learn how to store data on the PicoW microcontroller. This will allow you to unplug the PicoW from your laptop, plug it into a charger and leave it to take data and store it.

> ***Warning: If you are not careful, you can fill up all of the free space on the microcontroller which may cause it to become unstable and possible lose all files on the microcontroller!***

One of the challenges I add for this chapter is finding a way to keep your program from using all of the free space.

### *Reading and Writing Files*

For any programming language, reading and writing files is an important part of most programming tasks.It is amazing to me that these little microcontrollers can use standard python code to do this.

- **Challenge**: Modify the temperature program to write the temperature of an external TMP36 temperature sensor.
- **Finding free space on the PicoW**. The code below prints out the number of free bytes on the PicoW.

```
import os
(blksize, fragsize, nblocks, nfree, _, _, _, _, _, _) = os.statvfs('/')
n_bytes_free =  nfree * blksize
print(f"There are {n_bytes_free} bytes are available.")
```

- **Challenge**: Modify your logging program so that is stops if the number of free bytes is less than 10,000 bytes free.