Fundamentals of Applied Microcontrollers Manual

Seth McNeill

Edition Spring 2023 (v0.5)
2023 February 09

Contents

Chapter 1

Introduction

1.1 Introduction

This book is the accompanying text to a class introducing microcontrollers to upper division, non-electrical engineering undergraduate students who have taken some C programming.

The accompanying laboratory manual is located here.

1.2 License

Chapter 2

Number Systems

This chapter introduces the concepts of different numbering systems that are relevant to microcontrollers.

2.1 Decimal Numbers

The standard number system used by humans is based on 10. This logically flows the usual numbers of fingers or toes humans have. When counting in the base 10 (decimal) we start at 0, count up to 9, then run out of numbers to use. When runs out of numbers, we put a number to the left of the column we were counting in, and then start over at zero again which gives us the number 10. A more clear way to count would be to count from 00 to 09, then increment the left digit and start the right digit back at 0. This can be continued until 99 is reached. But now we have a model to follow. If a column gets to 9, we increment a column to the left and start the current column over at 0. This leads to the number 100. This concept can be continued forever.

When given a particular number, 3254, in decimal the value is calculated as

$$3254 = 3 \cdot 1000 + 2 \cdot 100 + 5 \cdot 10 + 4 \tag{2.1}$$

It could also be represented as

$$3254 = 3 \cdot 10^3 + 2 \cdot 10^2 + 5 \cdot 10^1 + 4 \cdot 10^0 \tag{2.2}$$

This is the basis for a positional number system.

2.2 Binary Numbers

Computers run a base 2 system, also known as binary. This means that we only have two options at each position, 1 or 0, instead of the 10 available in the decimal system. However, counting is done in the same way, when you run out of symbols, increment the position to the left and then start over at zero in the current position.

Binary numbers are also positional so a number like 1011 is

$$1011 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 11_{10} \tag{2.3}$$

where 11_{10} indicates that the number 11 is base 10.

The maximum value a binary number can have is determined by the number of bits it has. The formula is:

$$V_{max} = 2^n - 1 \tag{2.4}$$

where n is the number of bits. For example, an 8 bit number has a maximum value of $2^8 - 1 = 255$.

2.3 Hexadecimal Numbers

The problem for humans is that we have a very difficult time reading binary numbers. Especially once the numbers get long such as 11101101010010110101_2. The first thing we can do to improve intelligibility is to put a space in every 4 digits so that we aren't completely bowled over by all the digits. This gives us $1110\ 1101\ 0100\ 1011\ 0101_2$. This is better but still leaves some challenges. Someone pointed out that since 4 bits can have 16 different values, what if each 4 bits (also called a nibble) was represented by a single base 16 number. Base 16 is called hexadecimal. It has the problem that it runs out of digits once we get to 10, so it was decided to simply start on the alphabet so 10_{10} is A_{16}, 11_{10} is B_{16}, and so forth. Now the cumbersome binary number we've been playing with can be represented as $ED4B5_{16}$.

A comparison of the different numbering system representations of 0 through 15_{10} are shown in Table 2.1.

Decimal	Binary	Hexadecimal
00	0000	0
01	0001	1
02	0010	2
03	0011	3
04	0100	4
05	0101	5
06	0110	6
07	0111	7
08	1000	8
09	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Table 2.1: Decimal, binary, and hexadecimal numbers align as shown in this table.

2.4 Binary Background

The reasons for this stem from early computers using switches (relays) as their basic computing elements and the fact that telling the difference between a "high" voltage and a "low" voltage is easier to do than to differentiate 10 different voltages. It also allows for a gap between high and low that creates some immunity to noise called the noise margin. The figure linked to labels noise margin at NM which indicates how much (in volts) the signal coming from the source on the left can be degraded and still be recognized as a high or low. The other cool thing about digital signals is that each device the signal passes through removes all noise.

2.5 Converting Between Bases

2.5.1 Binary to Decimal

Converting to decimal is straightforward since we can simply sum each digit multiplied by the power of 2 it represents.

$$V_{10} = \sum_{i=0}^{p} d_i \cdot 2^i \tag{2.5}$$

In Equation 2.5 the p digits, d_i, of the binary number are summed from right to left while being multiplied by the power of 2 they represent. As an example, convert 1011 0111 to decimal.

$$\begin{aligned} V_{10} &= 1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3 + 1 \cdot 2^4 + 1 \cdot 2^5 + 0 \cdot 2^6 + 1 \cdot 2^7 \\ &= 1 + 2 + 4 + 0 + 16 + 32 + 0 + 128 \\ &= 183 \end{aligned} \tag{2.6}$$

I usually find it easiest to just remember the powers of two for each place and add them up.

$$\overset{128}{1}\ \ \overset{64}{0}\ \ \overset{32}{1}\ \ \overset{16}{1}\ \ \overset{8}{0}\ \ \overset{4}{1}\ \ \overset{2}{1}\ \ \overset{1}{1} = 1 + 2 + 4 + 16 + 32 + 128 \tag{2.7}$$

2.5.2 Decimal to Binary

Converting decimal numbers to binary involves dividing by 2 until the remainder is 0 or 1. The process goes as follows:

1. Divide the number by 2. If the remainder is 1 then the least significant bit is 1 otherwise if the remainder is 0 (it was an even number) then the least significant bit is 0.
2. Take the quotient from the previous step and divide by 2. The remainder is the next more significant bit in the binary representation.
3. Repeat Step 2 until the quotient is 0.

As an example, let's convert 11_{10} to binary.

$$\begin{aligned} 11 \div 2 &= 5R1 && \rightarrow 1 \text{ (LSB)} \\ 5 \div 2 &= 2R1 && \rightarrow 1 \\ 2 \div 2 &= 1R0 && \rightarrow 0 \\ 1 \div 2 &= 0R1 && \rightarrow 1 \text{ (MSB)} \end{aligned} \tag{2.8}$$

That means that 11_{10} is 1011_2. As another example let us convert 20_{10} to binary.

$$\begin{aligned}
20 \div 2 &= 10R0 && \rightarrow 0 \text{ (LSB)} \\
10 \div 2 &= 5R0 && \rightarrow 0 \\
5 \div 2 &= 2R1 && \rightarrow 1 \\
2 \div 2 &= 1R0 && \rightarrow 0 \\
1 \div 2 &= 0R1 && \rightarrow 1 \text{ (MSB)}
\end{aligned} \tag{2.9}$$

This shows that 20_{10} is 10100_2.

2.5.3 Binary and Hexadecimal

For conversions between binary and hexadecimal I tend to use the table lookup method. After using it enough times you begin to memorize the conversions. In my head I'm usually converting from binary to decimal on each nibble and then converting decimal into hexadecimal. So if I see 0101 I remember it is 5 in decimal which is the same in hex. If I see 1010 I remember that it is $8 + 2 = 10$ which is one more than 9 so it is A in hex.

2.6 Colors

Colors for display on computers are represented as binary numbers. Colors are 24 bit which is broken down into 3 8 bit numbers representing red, green, and blue (RGB). Eight bits give values over the range of 0 to 255 so colors are represented by 3 numbers such as (255, 0, 255) which gives a color like this. It is good to note that (255, 255, 255) is white, (0, 0, 0) is black, and (X, X, X) where all three numbers are the same is gray.

Sometimes colors are represented as a 6 digit binary number in hexadecimal form prefixed by 0x such as 0xFF00FF for the color we looked at previously.

2.7 ASCII

Since computers operate using binary numbers, how do we get them to represent human languages? The method caught it is called American Standard Code for Information Interchange (ASCII). Basically, 7 bit numbers were mapped to letters, numbers, punctuation, symbols, and control characters. Some examples are A is 65, Z is 90, a is 97, z is 122, 0 is 48, 9 is 57. The most used control characters are carriage return (13 or \r) and line feed (10 or \n). Unix based operating systems (Linux, OS X) use line feed to indicate a new line. Windows uses both (\r\n).

Look up an ASCII table online when you need to know the values.

2.8 Adding and Subtracting Binary Numbers

2.8.1 Default method

2.8.1.1 Addition

Calculate the binary sum 0001 0011 1101 + 0000 1011 0111.

$$\begin{array}{ccccccccccccc}
 & 0 & 0 & 0 & 1 & 0 & \overset{1}{0} & \overset{1}{1} & \overset{1}{1} & \overset{1}{1} & \overset{1}{1} & \overset{1}{0} & 1 \\
+ & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\
\hline
 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0
\end{array} \tag{2.10}$$

2.8.1.2 Subtraction

Calculate the binary difference 0001 0011 1101 − 0000 1011 0111.

$$\begin{array}{ccccccccccccc}
 & 0 & 0 & 0 & \overset{0}{\cancel{1}} & \overset{10}{\cancel{0}} & 0 & 1 & 1 & \overset{0}{\cancel{1}} & \overset{\overset{10}{\cancel{0}}}{\cancel{1}} & \overset{10}{\cancel{0}} & 1 \\
- & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\
\hline
 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0
\end{array} \tag{2.11}$$

2.8.2 Twos Complement method

In order to represent negative numbers a set of logic has to be overlaid over the basic unsigned integer binary number system. One method is called the Two's Complement method. This arranges 4-bit binary numbers as shown in Table 2.2. The great part about the two's complement method is that adding and subtracting can be done with no knowledge of whether the numbers involved are two's complement or not. Adding and subtracting can proceed as if the numbers are unsigned and then the interpretation as a negative or positive number can be done afterwards. Examples are shown in Equations 2.12 and 2.13.

Decimal	Two's Complement
7	0111
6	0110
5	0101
4	0100
3	0011
2	0010
1	0001
0	0000
−1	1111
−2	1110
−3	1101
−4	1100
−5	1011
−6	1010
−7	1001
−8	1000

Table 2.2: This table shows the decimal and two's complement numbers for 4-bits.

Calculate the two's complement binary sum $1101 + 0011$ $(-3 + 3 = 0)$.

$$\begin{array}{rcccc} & \overset{1}{1} & \overset{1}{1} & \overset{1}{0} & 1 \\ + & 0 & 0 & 1 & 1 \\ \hline & 0 & 0 & 0 & 0 \end{array} \tag{2.12}$$

Calculate the two's complement binary difference $1101 - 0011$ $(-3 - 3 = -6)$.

$$\begin{array}{rcccc} & 1 & \overset{0}{\cancel{1}} & \overset{10}{\cancel{0}} & 1 \\ - & 0 & 0 & 1 & 1 \\ \hline & 1 & 0 & 1 & 0 \end{array} \tag{2.13}$$

2.9 Gray Codes

When counting in binary, often times more than one bit changes as the number increments. This works fine in the ideal world, but in the real world, the logic controlling each bit might be slightly different. This will cause one bit to change at a slightly different time than another bit. That causes glitches in the counting that can be disruptive to the overall system.

Take as an example a robot that represents the cardinal directions as binary integers as illustrated in Figure 2.1.

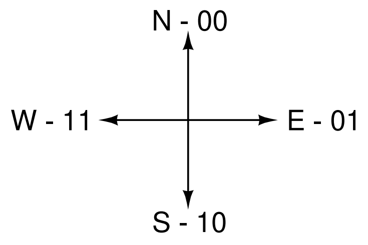

Figure 2.1: This figure shows using regular binary counting to represent the cardinal directions.

Using regular binary counting means that as the direction changes from East to South, two bits have to change. If the bits are driven by real switches or differing logic they may not change simultaneously leading to possible outputs of E - W - S or E - N - S. If the data is being used in a sequential manner it could lead to erroneous actions by the device (robot, car, etc.). Instead of using regular binary counting, we could use an encoding that only changes one bit at a time as the directions change as shown in Figure 2.2.

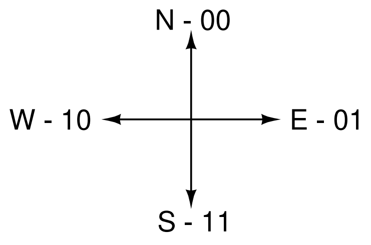

Figure 2.2: This figure shows using Gray codes to represent the cardinal directions.

Counting systems like this are called Gray code after Frank Gray or reflected binary code. A 4-bit example is shown in Table 2.3.

Gray Code
0000
0001
0011
0010
0110
0111
0101
0100
1100
1101
1111
1110
1010
1011
1001
1000

Table 2.3: This table shows 4-bit Gray codes with horizontal lines showing the break for 2- and 3-bit Gray codes.

2.10 Binary Background

Originally, computers were just a bunch of switches, therefore, they only had two positions: on and off. This also has the benefit that there is a large amount of noise immunity. A repeater (buffer) can eliminate by recreating the original signal without noise. These are all great benefits of the binary system.

Chapter 3

Boolean Logic

3.1 Introduction

This chapter introduces some basic Boolean logic including gates and Boolean algebra.

When can you drive through an intersection? When the light is NOT red. This is the first and simplest logic operator–the NOT element. It simply changes any TRUE to FALSE or FALSE to TRUE (you can substitute 1 for TRUE and 0 for FALSE, or on/off).

How do you start a car? In most of the cars I have driven I have to press on the brake at the same time as I turn the key. To say it another way, the car starts when I press the break AND turn the key.

$$pressBreak\,\mathrm{AND}\,turnKey = startedCar \tag{3.1}$$

Again on a car, a particular blinker light will turn on if you turn on the turn signal or if you turn on the 4-way blinker.

$$turnSignal\,\mathrm{OR}\,4wayBlinker = blinking \tag{3.2}$$

The AND and OR in the equation are Boolean operators.

3.2 Methods of Representing Logic

It is important to differentiate between different forms of representation because $\overline{\mathrm{EN}}$ means **active low** enable. It does not mean NOT(E AND N). The understanding of which method is being used is usually derived from context. Using the dot $(\cdot)$ everywhere can become burdensome, so when context makes it obvious (usually examples involving A, B, and C or X and Y) we may drop the dots and just use adjacency to represent the AND function.

In this class, most of the logic you write will be done in C++. Some of the following tables will also address how to represent logic in C++.

3.3 Boolean Algebra

3.3.1 Theorems of Boolean Algebra

Ways to show NOT:

$$\begin{array}{rll} \text{Algebra} & \text{C}++ & \\ NOT(X) = \overline{X} = X' & !x & (3.3a) \\ \overline{(\overline{X})} = (X')' = X & !(!x) = x & (3.3b) \end{array}$$

Rules of AND and OR:

$$\begin{array}{rll} \text{AND} & \text{OR} & \\ 0 \cdot 0 = 0 & 1 + 1 = 1 & (3.4a) \\ 1 \cdot 1 = 1 & 0 + 0 = 0 & (3.4b) \\ 0 \cdot 1 = 1 \cdot 0 = 0 & 1 + 0 = 0 + 1 = 1 & (3.4c) \end{array}$$

And repeated in C++ form:

$$\begin{array}{rll} \text{AND} & \text{OR} & \\ \text{false \&\& false == false} & \text{true || true == true} & (3.5a) \\ \text{true \&\& true == true} & \text{false || false == false} & (3.5b) \\ \text{true \&\& false == false} & \text{true || false == true} & (3.5c) \end{array}$$

$$\begin{array}{rll} \text{AND} & \text{OR} & \\ X \cdot 1 = X & X + 0 = X & (3.6a) \\ X \cdot 0 = 0 & X + 1 = 1 & (3.6b) \\ X \cdot X = X & X + X = X & (3.6c) \\ X \cdot \overline{X} = 0 & X + \overline{X} = 1 & (3.6d) \end{array}$$

For two and three variables you have the following useful equations:

$$\begin{array}{rll}
\text{AND} & \text{OR} & \\
X \cdot Y = Y \cdot X & X + Y = Y + X & (3.7a) \\
(X \cdot Y) \cdot Z = X \cdot (Y \cdot Z) & (X + Y) + Z = X + (Y + Z) & (3.7b) \\
(X + Y) \cdot (X + Z) = X + Y \cdot Z & X \cdot Y + X \cdot Z = X \cdot (Y + Z) & (3.7c) \\
X \cdot (X + Y) = X & X + X \cdot Y = X & (3.7d) \\
(X + Y) \cdot (X + \overline{Y}) = X & X \cdot Y + X \cdot \overline{Y} = X & (3.7e) \\
X \cdot (\overline{X} + Y) = X \cdot Y & X + \overline{X} \cdot Y = X + Y & (3.7f)
\end{array}$$

$$X \cdot Y + \overline{X} \cdot Z + Y \cdot Z = X \cdot Y + \overline{X} \cdot Z \qquad (3.7g)$$

$$(X + Y) \cdot (\overline{X} + Z) \cdot (Y + Z) = (X + Y) \cdot (\overline{X} + Z) \qquad (3.7h)$$

A very important pair of equations are DeMorgan's theorems which allow us to switch between sums of products and products of sums.

$$\overline{(X_1 \cdot X_2 \cdot \ \cdots \ \cdot X_n)} = \overline{X_1} + \overline{X_2} + \cdots + \overline{X_n} \qquad (3.8a)$$

$$\overline{(X_1 + X_2 + \cdots + X_n)} = \overline{X_1} \cdot \overline{X_2} \cdot \ \cdots \ \cdot \overline{X_n} \qquad (3.8b)$$

The following are very important to remember:

$$\overline{A} \cdot \overline{B} \neq \overline{AB} \qquad (3.9a)$$

$$\overline{A} + \overline{B} \neq \overline{A + B} \qquad (3.9b)$$

3.4 Logic Gates and Truth Tables

It is important to be able to transform between equation, diagrams/circuits, and truth tables.

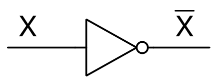

Figure 3.1: The NOT gate, also called an inverter, outputs NOT(A).

A	X
0	1
1	0

Table 3.1: This is the truth table for a NOT gate.

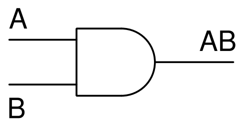

Figure 3.2: The AND gate outputs A AND B.

A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

Table 3.2: This is the truth table for an AND gate.

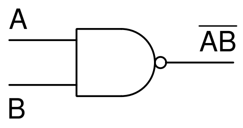

Figure 3.3: The NAND gate outputs NOT(A AND B).

A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

Table 3.3: This is the truth table for an NAND gate.

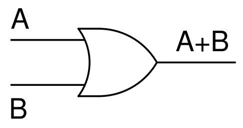

Figure 3.4: The OR gate outputs A OR B.

A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

Table 3.4: This is the truth table for an OR gate.

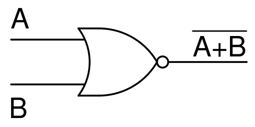

Figure 3.5: The NOR gate outputs NOT(A OR B).

A	B	X
0	0	1
0	1	0
1	0	0
1	1	0

Table 3.5: This is the truth table for an NOR gate.

Chapter 4

Arduino Startup

4.1 Introduction

This chapter gives the students an introduction to the hardware we are using and gets them started with the Arduino IDE.

4.2 Datasheets

4.2.1 Arduino Nano Connect RP2040

The data sheet for the Arduino Nano Connect RP2040 is located at
https://docs.arduino.cc/resources/datasheets/ABX00053-datasheet.pdf.

The main website for it is at
https://docs.arduino.cc/hardware/nano-rp2040-connect

The pinout is at
https://content.arduino.cc/assets/Pinout_NanoRP2040_latest.png.

4.2.1.1 RP2040 Microcontroller

The RP2040 microcontroller datasheet (all 654 pages) is at
https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf.

If you are ever interested in putting the microcontroller onto a circuit board yourself, there is a reference design at
https://datasheets.raspberrypi.com/rp2040/hardware-design-with-rp2040.pdf.

4.2.1.2 IMU - ST LSM6DSOXTR

The IMU datasheet is at
https://www.st.com/resource/en/datasheet/lsm6dsox.pdf

4.2.1.3 Mic - ST MP34DT06JTR

The microphone datasheet is at
https://www.st.com/resource/en/datasheet/mp34dt06j.pdf

The overview page shows that the microphone is still actively being produced.

4.2.1.4 WiFi and Bluetooth - U-blox® Nina W102

The main page for the wireless unit is at
https://www.u-blox.com/en/product/nina-w10-series-open-cpu.

The datasheet is at
https://www.u-blox.com/sites/default/files/NINA-W10_DataSheet_UBX-17065507.pdf.

4.2.1.5 Cryptographic IC - Microchip® ATECC608A

Note that Microchip doesn't suggest using this chip in new designs so expect that some future versions of the Nano RP2040 Connect to use the successor (ATTECC608B).

The datasheet for the 608A is here:
https://ww1.microchip.com/downloads/en/DeviceDoc/ATECC608A-CryptoAuthentication-Device-Summary-Data-Sheet-DS40001977B.pdf.

Note that it says it is a summary datasheet. You have to sign an NDA with Microchip to see the full datasheet.

4.2.2 Circuit Board Parts (v0.5)

1. DS18B20+ 1-Wire temperature sensor
2. GX18B20 1-Wire temperature sensor
3. VL6180 Distance sensor
4. SHT31 Temperature and Humidity sensor
5. TC1047 Analog out temperature sensor

6. ADS7142 Analog-to-digital converter (ADC)
7. Adafruit 1.14” 240x135 Color TFT Display + MicroSD Card Breakout - ST7789
8. U3V40FX Voltage regulator
9. TMI8837 Motor controller
10. QMC5883L Compass
11. ST25DV16 NFC I2C
12. NeoPixels (WS2812)
13. PCA9535 I2C GPIO
14. SLR0394 LED display

4.3 Schematics and PCB

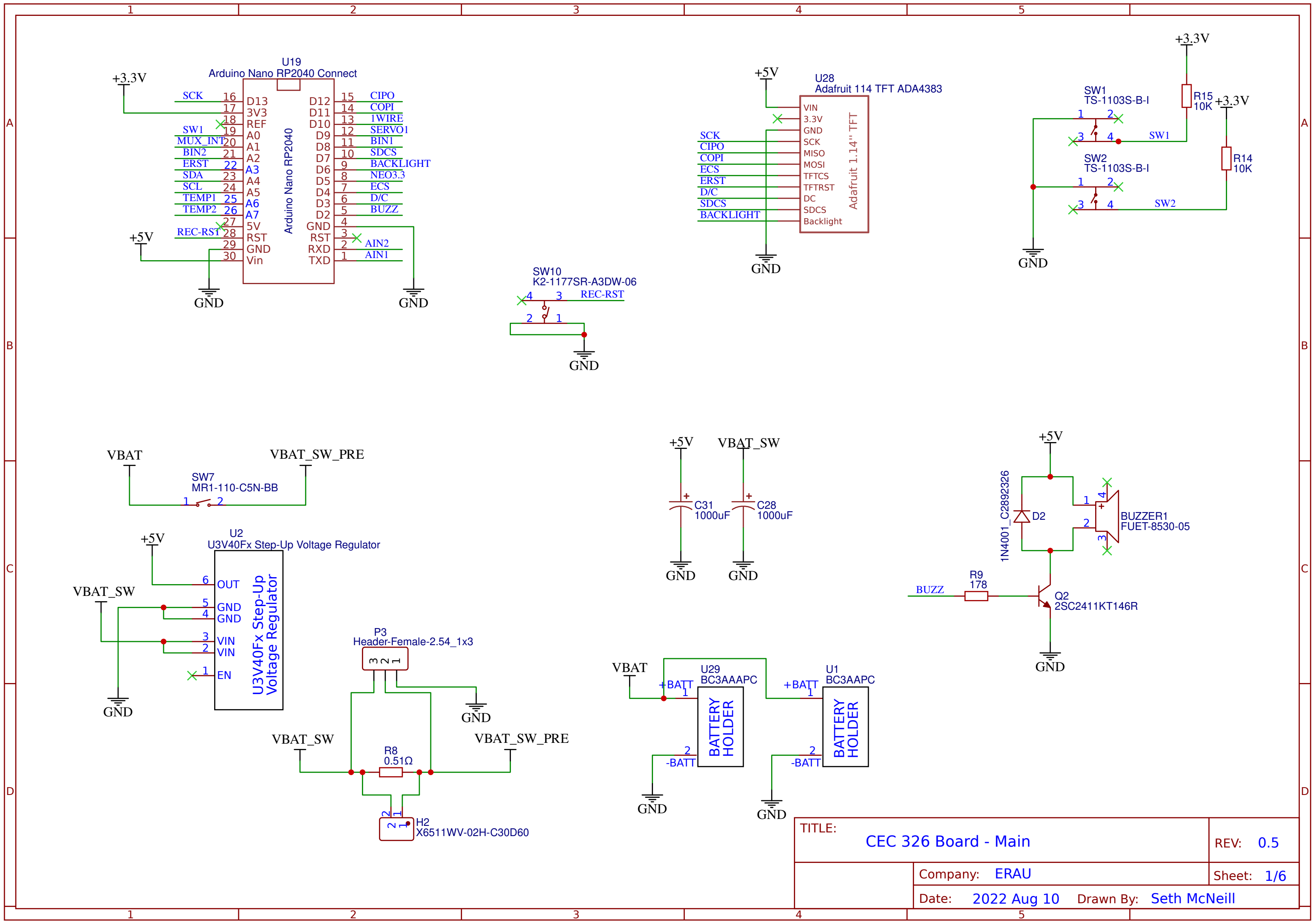

Figure 4.1: This is the schematic of the version 0.5 of the board. This is the board used in Fall 2022.

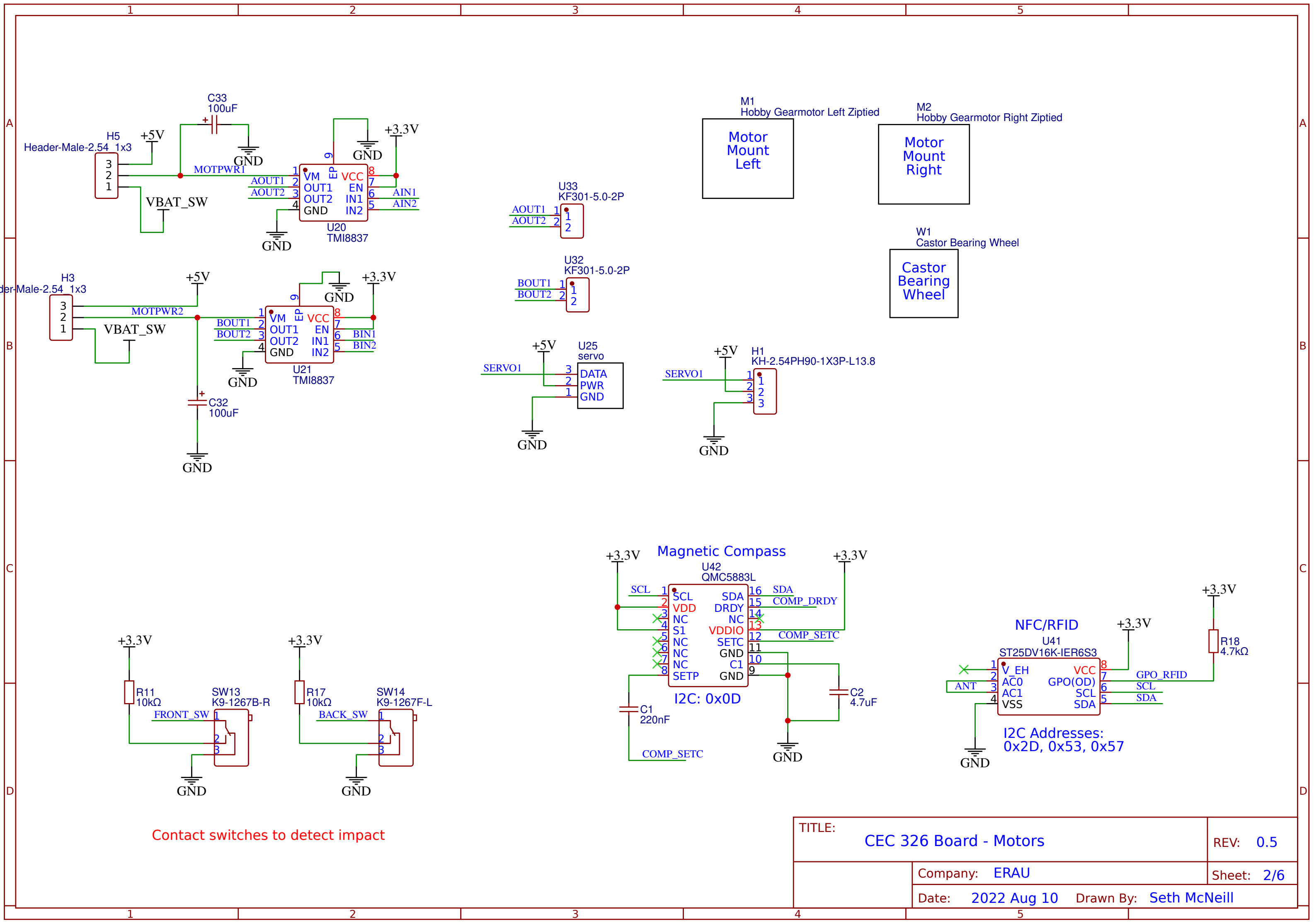

Figure 4.2: This is the schematic of version 0.5 of the board. This is the board used in Fall 2022.

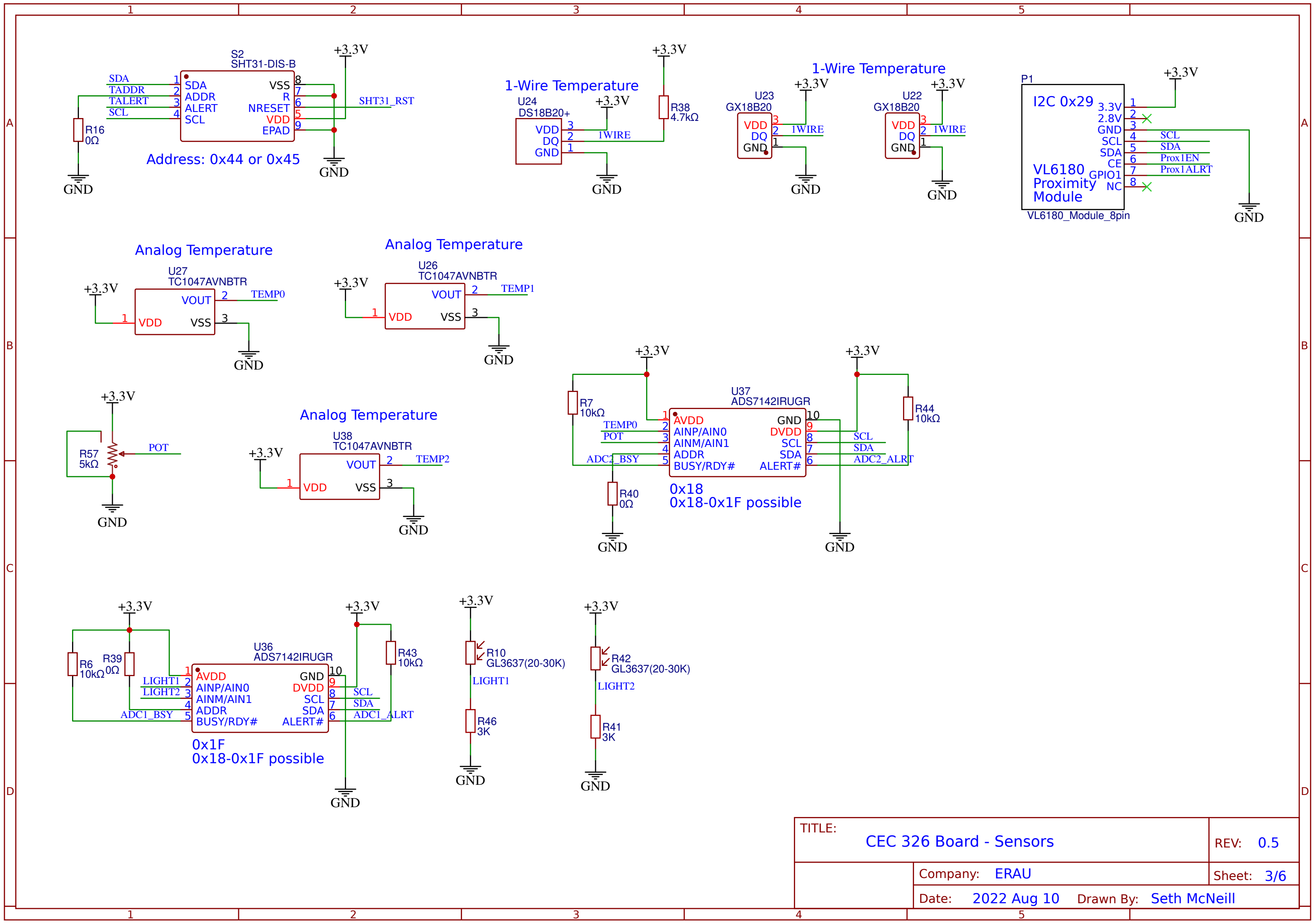

Figure 4.3: This is the schematic of version 0.5 of the board. This is the board used in Fall 2022.

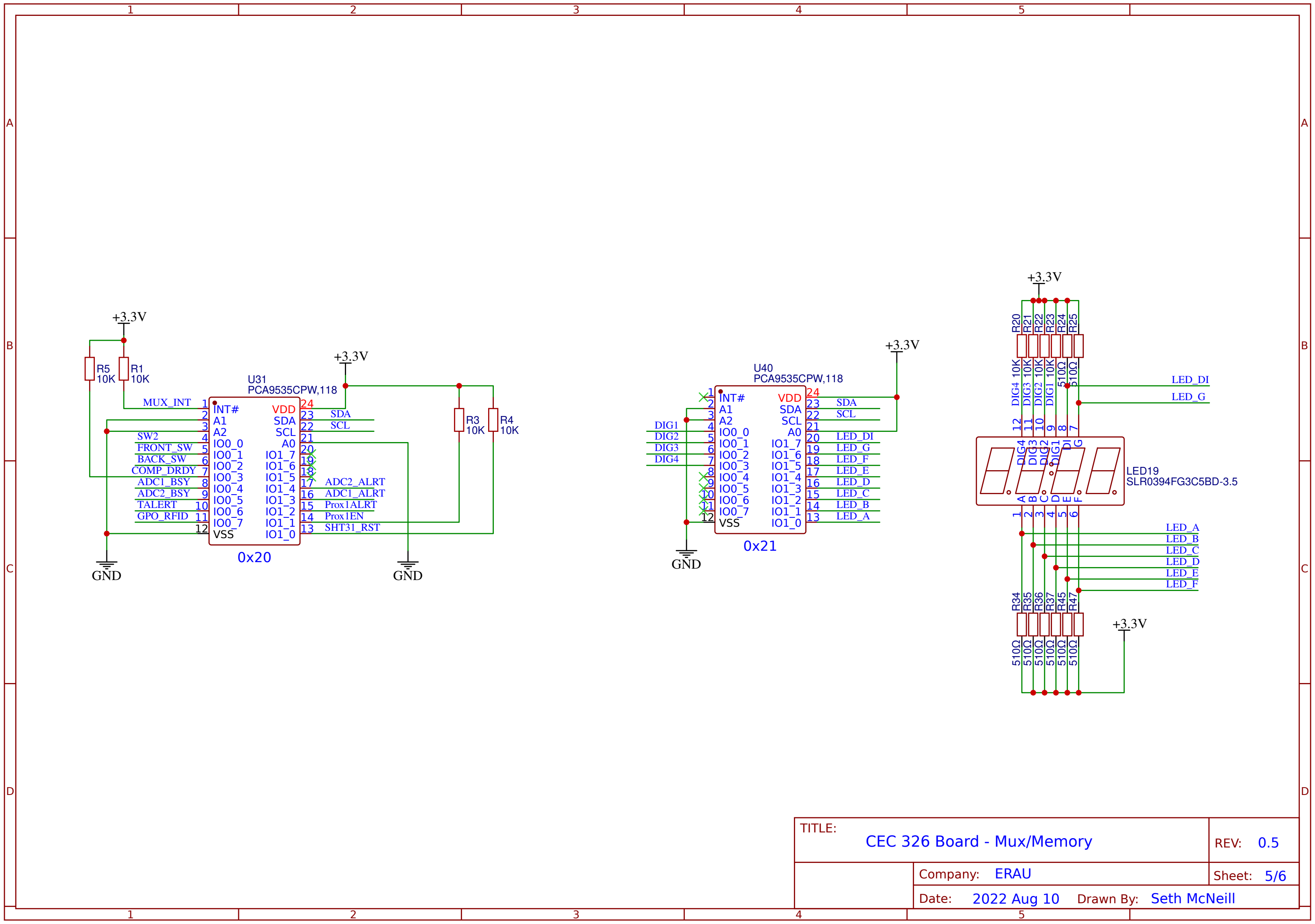

Figure 4.4: This is the schematic of version 0.5 of the board. This is the board used in Fall 2022.

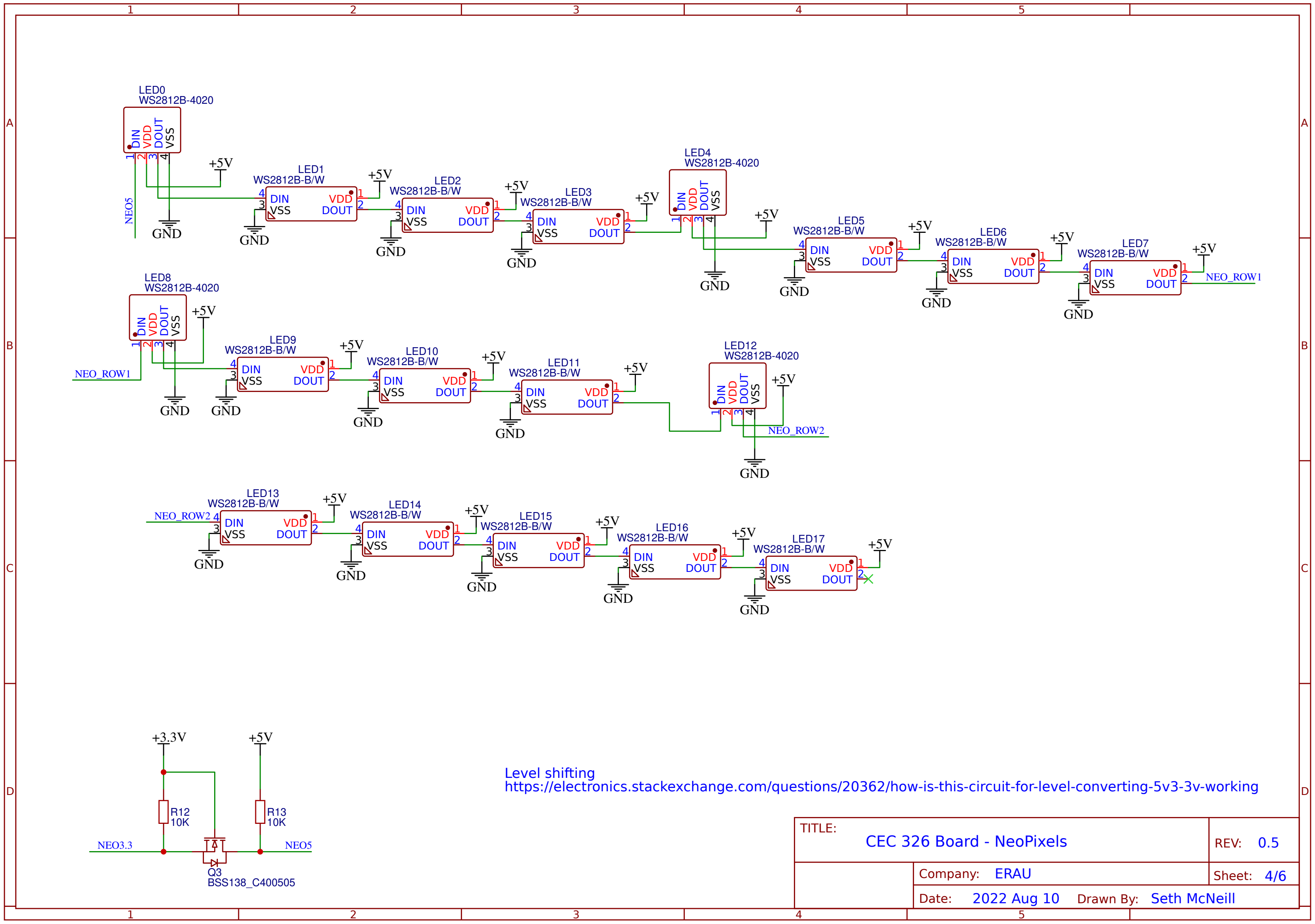

Figure 4.5: This is the schematic of version 0.5 of the board. This is the board used in Fall 2022.

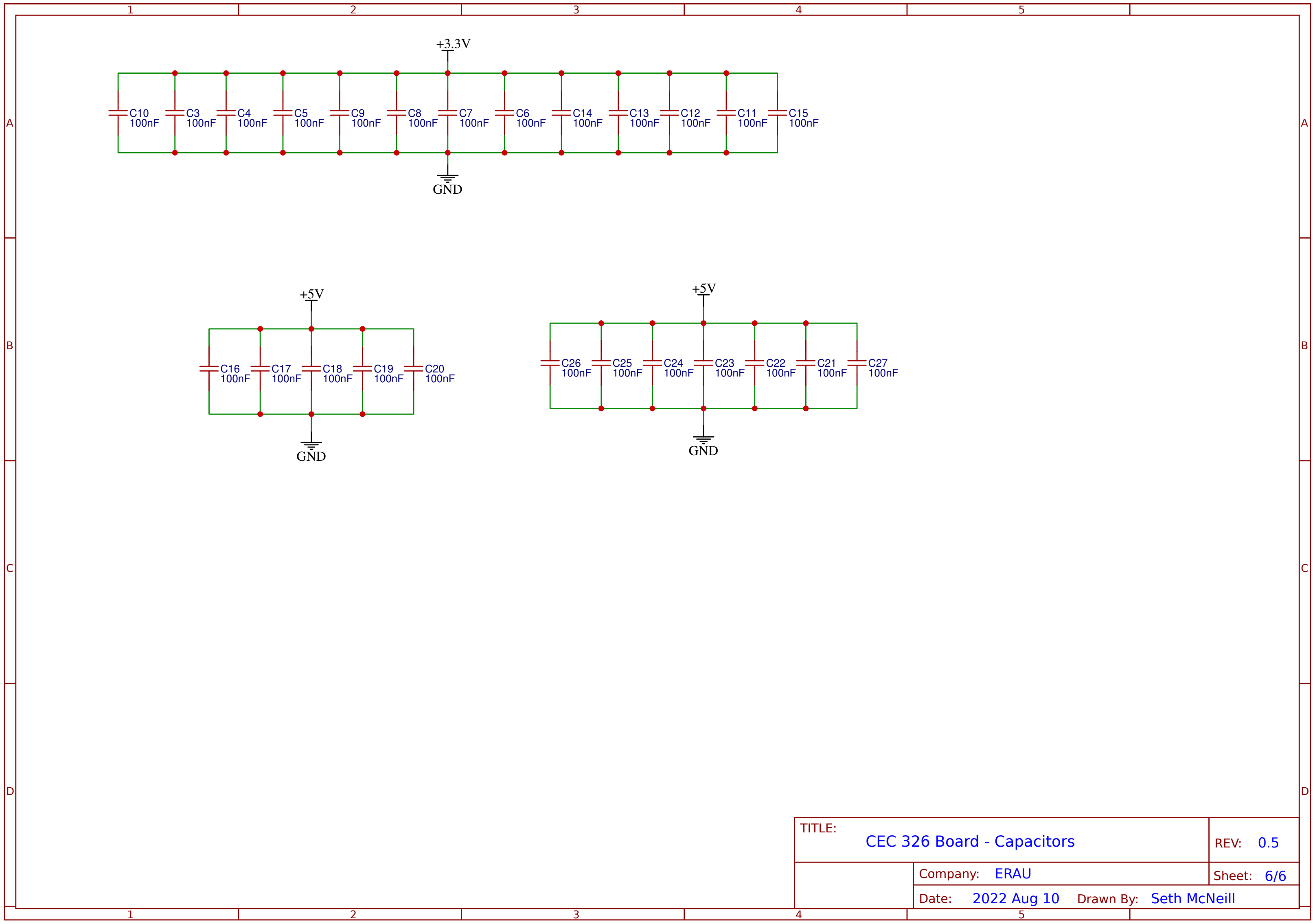

Figure 4.6: This is the schematic of version 0.5 of the board. This is the board used in Fall 2022.

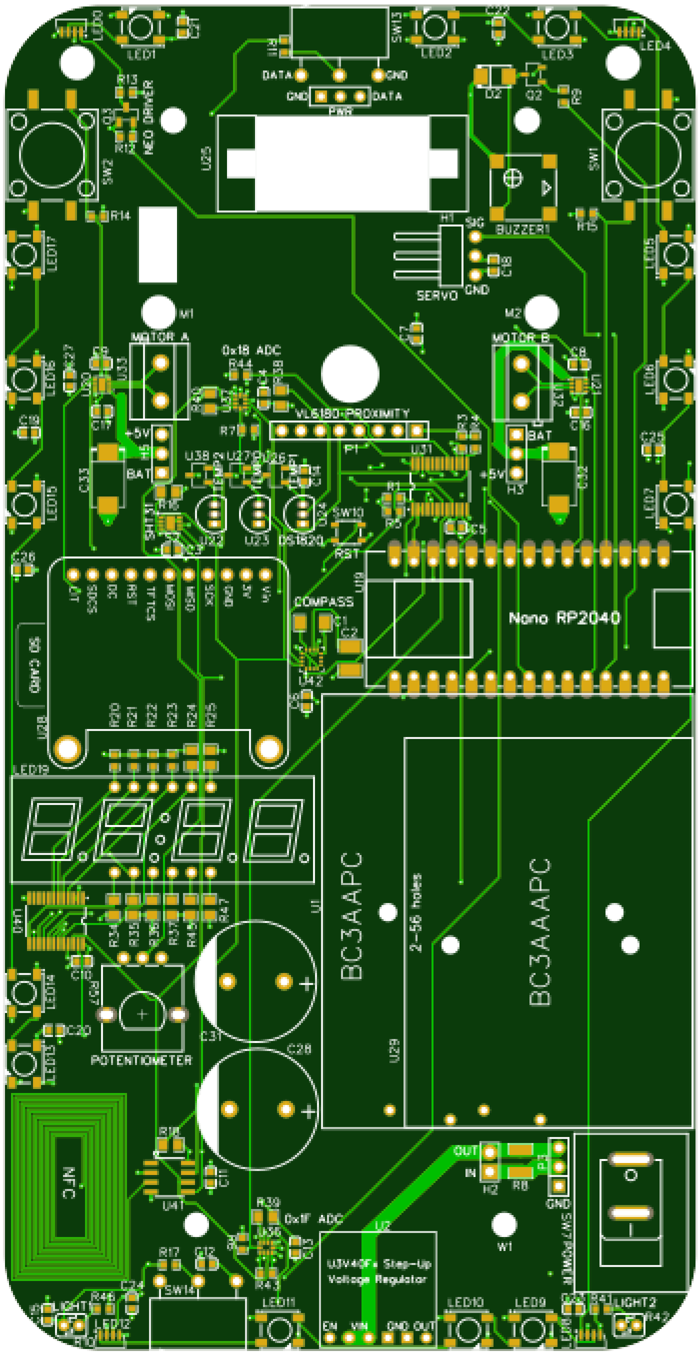

Figure 4.7: This shows the top of version 0.5 of the board for this class to help locate components.

Chapter 5

I2C GPIO/Multiplexer

5.1 Introduction

This chapter introduces students to using the PCA9535 I^{2}C GPIO chip to control LEDs and input button presses. The PCA9535 has 16 I/O pins that are controlled via I^{2}C. It's I^{2}C address can be set between 0x20 and 0x27 through setting the A0-A2 pins either high (1) or low (0). These pins control the lower 3 bits of the address. The two 8-bit I/O pin ports (IO0 and IO1) are controlled via a set of eight registers. Registers are binary numbers where each bit sets the configuration of one of the I/O pins.

5.2 Registers

5.2.1 Registers 0 and 1 - Input Registers

These registers reflect the incoming logic levels of the port pins no matter whether the port is configured as an input or an output. These ports are read only; you cannot write to them. Table 5.1 shows the registers.

5.2.2 Registers 2 and 3 - Output Registers

These registers control the output value of the pins that are set as outputs. Pins are set as outputs or inputs in Registers 6 and 7. Note that the default output value of all the pins is 1 or logic high. It is important to note that

I/O Input Register Port 0

Bit	7	6	5	4	3	2	1	0
Symbol	I0.7	I0.6	I0.5	I0.4	I0.3	I0.2	I0.1	I0.0
Default	X	X	X	X	X	X	X	X

I/O Input Register Port 1

Bit	7	6	5	4	3	2	1	0
Symbol	I1.7	I1.6	I1.5	I1.4	I1.3	I1.2	I1.1	I1.0
Default	X	X	X	X	X	X	X	X

Table 5.1: PCA9535 input port values are shown here. The Default value of X means that it's value is not set at power on. Also, note that the Port 0 symbols are the letter I followed by the number zero, not the letter O.

the values in these registers have no effect on pins that are defined as inputs. Table 5.2 shows the registers.

I/O Output Register Port 0

Bit	7	6	5	4	3	2	1	0
Symbol	O0.7	O0.6	O0.5	O0.4	O0.3	O0.2	O0.1	O0.0
Default	1	1	1	1	1	1	1	1

I/O Output Register Port 1

Bit	7	6	5	4	3	2	1	0
Symbol	O1.7	O1.6	O1.5	O1.4	O1.3	O1.2	O1.1	O1.0
Default	1	1	1	1	1	1	1	1

Table 5.2: PCA9535 output port values are shown here. The Default value of 1 means that it's has a high output at power on.

5.2.3 Registers 4 and 5 - Polarity Inversion Registers

For this class we will not be changing the Polarity Inversion Registers.

5.2.4 Registers 6 and 7 - Configuration Registers

These registers set whether a particular I/O is an input or an output. This is set on a per-pin basis. Setting a register bit to 1 configures the corresponding pin to be an input. Setting a register bit to a 0 configures the corresponding pin to be an output. Table 5.3 shows the registers.

I/O Configuration Port 0

Bit	7	6	5	4	3	2	1	0
Symbol	C0.7	C0.6	C0.5	C0.4	C0.3	C0.2	C0.1	C0.0
Default	1	1	1	1	1	1	1	1

I/O Configuration Port 1

Bit	7	6	5	4	3	2	1	0
Symbol	C1.7	C1.6	C1.5	C1.4	C1.3	C1.2	C1.1	C1.0
Default	1	1	1	1	1	1	1	1

Table 5.3: PCA9535 configuration port default values are shown here. Since all configuration bits default to a value of 1, all pins default to being inputs.

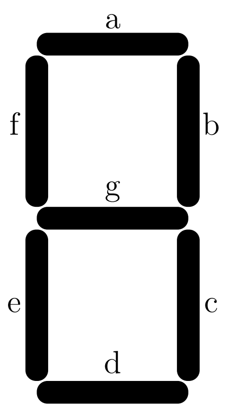

Figure 5.1: Seven segment LED display segments are labeled as shown.

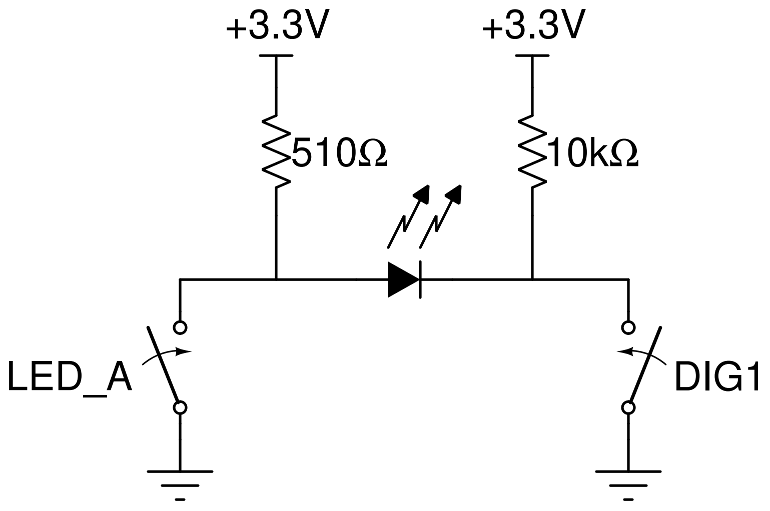

Figure 5.2: A single segment of a seven segment digit is connected with this circuit. Setting LED_A to 1 (open) and DIG1 to 0 (closed) lights up the segment.

Chapter 6

Buttons and Serial Communications

6.1 Introduction

This chapter introduces students to using buttons and serial communications.

6.2 Buttons

A typical button circuit is shown in Figure 6.1. The input is high until the button is pressed. Unfortunately, being mechanical, buttons do not always create nice, clean switched inputs as is shown in Figure 6.2. Maxim Integrated has a nice paper on the topic advertizing their devices to solve the problem. However, if we don't have the option of adding their hardware, some work can be done in software to debounce an input. As Maxim mentions, software debouncing is not free. It does incur some overhead so you probably do not want to do it for many inputs. An example of software debouncing is in Listing 6.1.

```
/*
Debounce

Each time the input pin goes from LOW to HIGH (e.g.
 because of a push-button
press), the output pin is toggled from LOW to HIGH
 or HIGH to LOW. There's a
```

```
minimum delay between toggles to debounce the
 circuit (i.e. to ignore noise).

The circuit:
- LED attached from pin 13 to ground through 220 ohm
  resistor
- pushbutton attached from pin 2 to +5V
- 10 kilohm resistor attached from pin 2 to ground

- Note: On most Arduino boards, there is already an
 LED on the board connected
  to pin 13, so you don't need any extra components
 for this example.

created 21 Nov 2006
by David A. Mellis
modified 30 Aug 2011
by Limor Fried
modified 28 Dec 2012
by Mike Walters
modified 30 Aug 2016
by Arturo Guadalupi

This example code is in the public domain.

https://www.arduino.cc/en/Tutorial/BuiltInExamples/
 Debounce
*/

// constants won't change. They're used here to set
 pin numbers:
const int buttonPin = 9;     // the number of the
 pushbutton pin
const int ledPin = 13;       // the number of the LED
  pin

// Variables will change:
```

```
int ledState = LOW;         // the current state of
 the output pin
int buttonState;             // the current reading
 from the input pin
int lastButtonState = LOW;   // the previous reading
  from the input pin

// the following variables are unsigned longs
 because the time, measured in
// milliseconds, will quickly become a bigger number
  than can be stored in an int.
unsigned long lastDebounceTime = 0;  // the last
 time the output pin was toggled
unsigned long debounceDelay = 50;    // the debounce
  time; increase if the output flickers

void setup() {
Serial.begin(115200);
while(!Serial) delay(10);
Serial.println("Starting...");

pinMode(buttonPin, INPUT);
pinMode(ledPin, OUTPUT);

// set initial LED state
digitalWrite(ledPin, ledState);
}

void loop() {
// read the state of the switch into a local
 variable:
int reading = digitalRead(buttonPin);

// check to see if you just pressed the button
// (i.e. the input went from LOW to HIGH), and you'
 ve waited long enough
// since the last press to ignore any noise:
```

```
// If the switch changed, due to noise or pressing:
if (reading != lastButtonState) {
  // reset the debouncing timer
  lastDebounceTime = millis();
}

if ((millis() - lastDebounceTime) > debounceDelay) {
  // whatever the reading is at, it's been there for
  longer than the debounce
  // delay, so take it as the actual current state:

  // if the button state has changed:
  if (reading != buttonState) {
  buttonState = reading;

  // only toggle the LED if the new button state is
 HIGH
  if (buttonState == HIGH) {
    ledState = !ledState;
  }
  }
}

// set the LED:
digitalWrite(ledPin, ledState);

// save the reading. Next time through the loop, it'
 ll be the lastButtonState:
lastButtonState = reading;

}
```

Listing 6.1: This is the Arduino example of software debouncing.

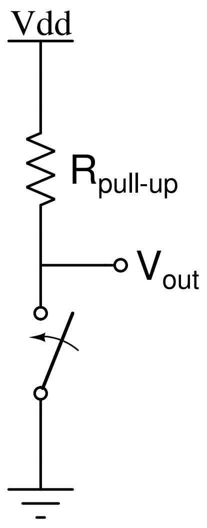

Figure 6.1: This is a typical button input circuit. It is active low in that the output signal will be high until the button is pressed.

6.3 Serial Communications

Serial communications sends data one bit at a time from one device to another as shown in Figure 6.3. This is in contrast to parallel communications where multiple bits (8 in the example) are transferred simultaneously between devices as shown in Figure 6.4. As can be seen parallel communications is much faster than serial since you can send so many bits simultaneously. However, parallel requires many more pins on each device to communicate. This is challenging in the embedded systems world since most microcontrollers don't have many pins. Also, connectors with many pins tend to fail more often than connectors with fewer pins. Because of these reasons, the embedded world communicates primarily with serial protocols.

A table of serial protocols is listed in Table 6.1.

6.3.1 Universal Asynchronous Receiver-Transmitter

The Universal Asynchronous Receiver-Transmitter (UART) protocol has been around quite a while. Older computers used to ship with serial ports that used this protocol, usually implemented as the RS-232 protocol. UARTs are used to communicate between two devices. It does not allow for more than

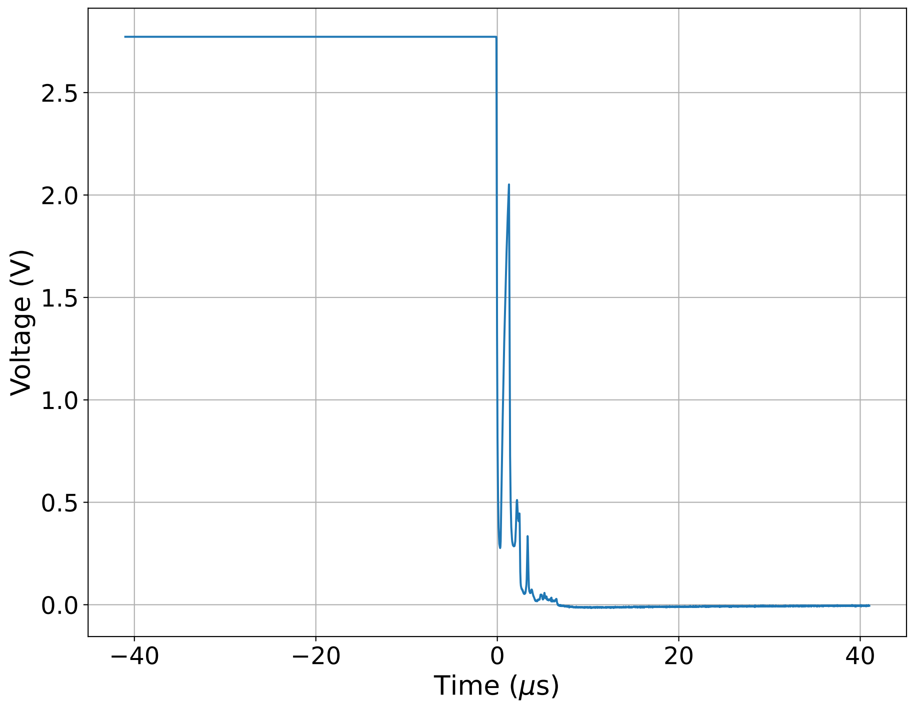

Figure 6.2: This is an example of what the output signal from a button with the circuit in Figure 6.1 could look like.

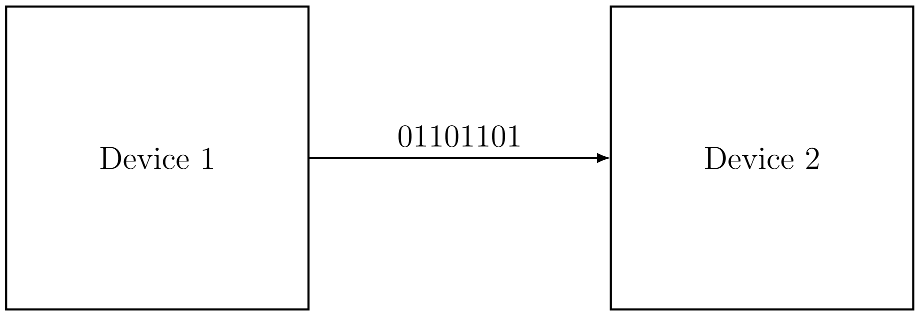

Figure 6.3: Serial transfers one bit at a time.

two devices. It is full duplex and communicates over 2 wires (plus a ground for reference). One wire transmits data from Device 1 to Device 2 and is

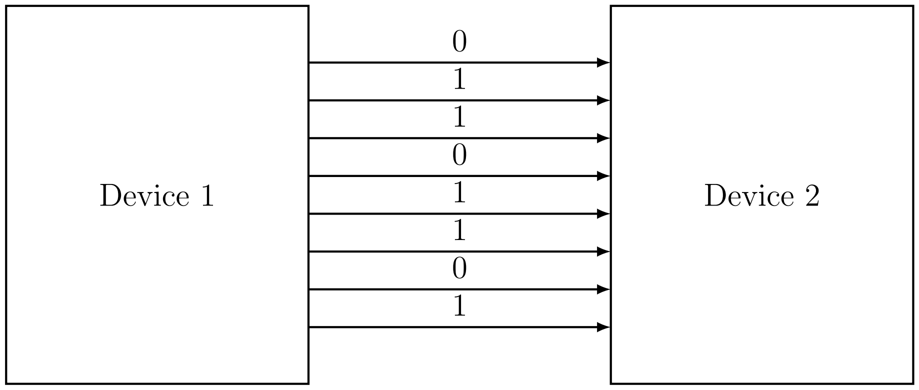

Figure 6.4: Parallel transfers multiple bits at a time.

Protocol	Description
UART	Used to communicate between computers and Arduino boards
SPI	Used for higher speed communications between devices on circuit boards
I^2C	Only uses two wires and allows for multiple controllers and peripherals
1-Wire	Only requires 1 wire and ground (no power) to communicate
CAN	*Often used in the automotive industry*
RS-485	*Differential signaling for robustness (noisy and long wires)*
USB	*Ubiquitous on computers now*

Table 6.1: This is a list of some of the more common serial protocols. The grayed out protocols will not be discussed further.

connected to Device 1's TX pin and Device 2's RX pin. The second wire transmits data from Device 2 to Device 1 and is connected to Device 1's RX pin and Device 2's TX pin. This is illustrated in Figure 6.5. This protocol does require both devices to use the same specified transmission speed. Some common speeds are 9600, 14,400, 57,600, and 115,200. It can go faster. Nowadays, it is generally best to go as fast as possible so that the communications takes less of the processor's time. The most common speed used now seems to be 115,200. The asynchronous part of the name comes from the fact that there is no clock line for UARTs. It typically has a maximum speed of 1.5 megabits per second (Mbps).

UART is the protocol used to communicate between the computer and your Arduino board. You can see this in the code in the setup function where `Serial.begin(115200)` or `Serial.begin(9600)` shows up often.

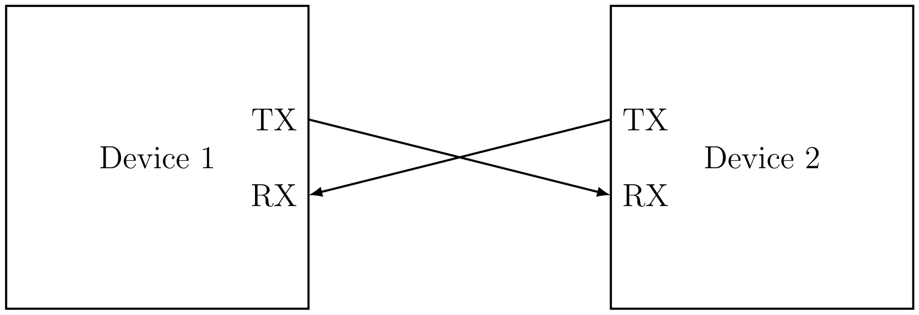

Figure 6.5: A UART has full duplex between two entities.

6.3.2 Serial Peripheral Interface

The Serial Peripheral Interface (SPI) is a full-duplex serial interface shown in Figure 6.6. It is setup in a controller-peripheral (previously known as master-slave) architecture with only one controller on the bus. It does allow for multiple peripherals by giving each peripheral its own chip select (CS) line. Since it is synchronous, it does require a clock line. This means that it takes a minimum of 4 wires as listed in Table 6.2. The SCLK signal is the clock signal used by both the controller and peripheral. There is one CS line per peripheral. The CS line lets the controller specify which peripheral it is communicating with. The COPI line is the data going from the controller to the peripheral. The CIPO line is the data going from the peripheral to the controller.

SPI has a maximum data rate of 60 Mbps, which is the fastest of the 3 protocols commonly used in the embedded world. Because of this, it is used in more data intense situations like SD Cards (where we will be using it) and displays.

Signal	Description
SCLK	Clock signal to keep everything synchronized
CS	Chip select–tells a peripheral that the controller is communicating with it
COPI	Controller Out, Peripheral In (used to be MOSI)
CIPO	Controller In, Peripheral Out (used to be MISO)

Table 6.2: The SPI protocol uses these signals to connect.

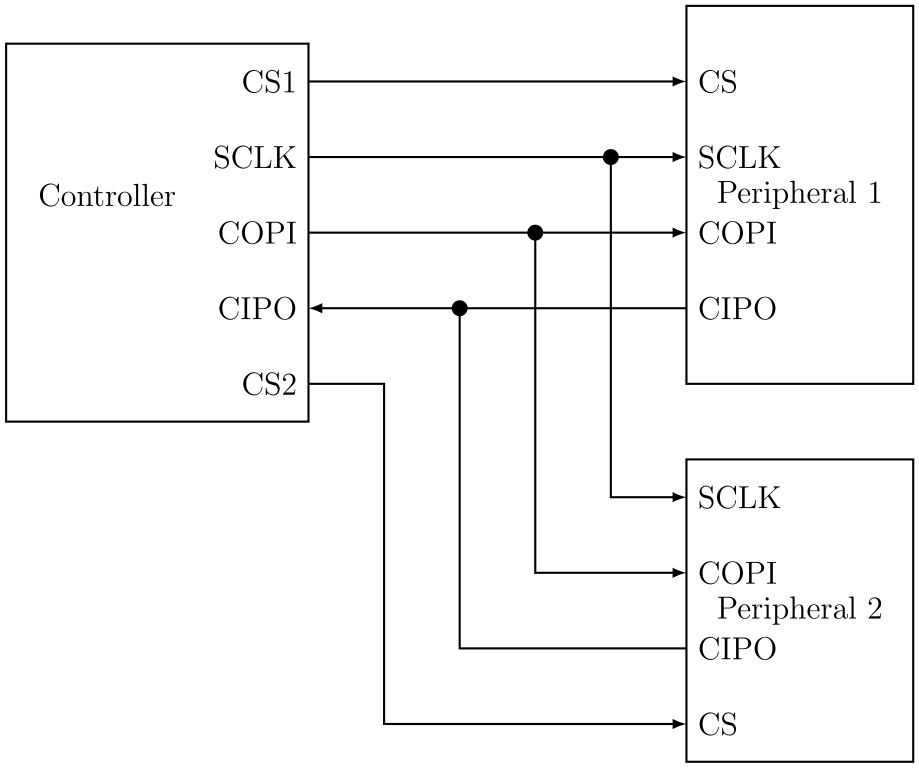

Figure 6.6: SPI allows for one (sometime more) controller and multiple peripherals.

6.3.3 Inter-Integrated Circuit

The Inter-Integrated Circuit (I^2C, IIC, or I2C) protocol is multi-controller and multi-peripheral. Basically, any device connected to the bus can drive the communication. It is single ended so data only flows one way at a time. It only requires 3 wires, SCL - clock, SDA - Data, and a ground reference. SCL and SDA do require pull-up resistors so that devices only have to pull the lines to ground to communicate as shown in Figure 6.7. Instead of using chip select lines like SPI, I2C requires each peripheral to have a unique address. I2C addresses for modules that are on the board or may be used with the board can be seen in Table 6.3. I2C has a max speed of 3.4 Mbps but is only 100 kbps in standard mode.

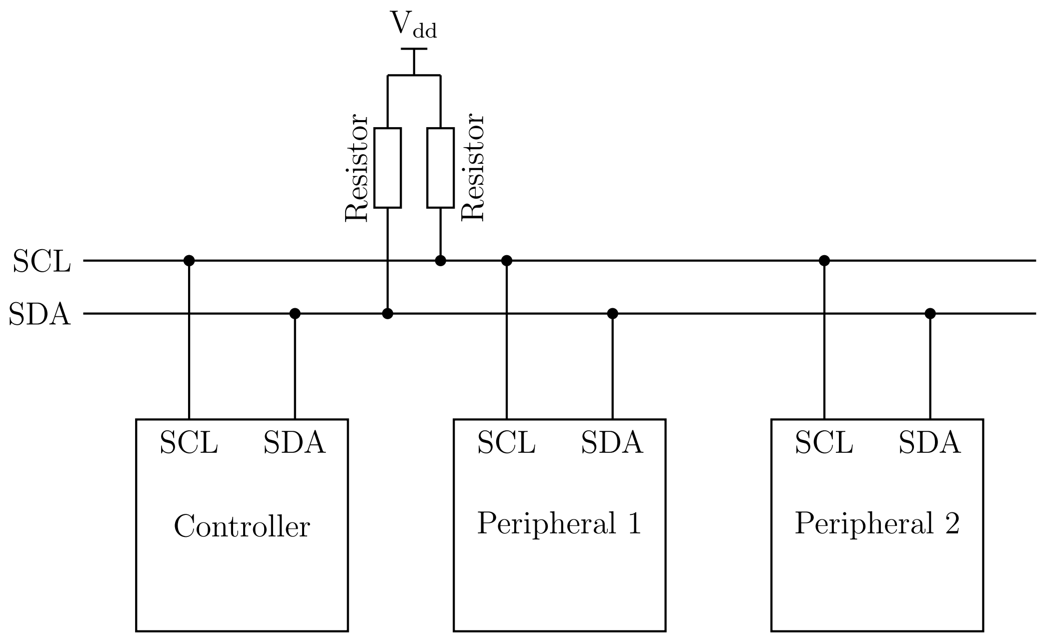

Figure 6.7: I^{2}C allows for multiple controllers and peripherals on the same bus.

Address (HEX)	Module
0x44 or 0x45	SHT31-DIS Temperature/Humidity
0x39	APDS-9960 Light, Color, Proximity, Gesture
0x77	BME688 Temperature, Humidity, Gas
0x2D, 0x53, and 0x57	ST25DV16 Dynamic NFC/RFID Tag IC
0x30 or other	NeoKey 1x4 QT breakout board
0x10	STEMMA MiniGPS
0x6A or 0x6B	LSM6DSOX IMU on the Nano Connect
0x60	ATECC608A Cryptographic on the Nano Connect
0x0D	QMC5883 Magnetic Compass
0x29	VL6180 Proximity Sensor, address moveable
0x18, 0x1F	ADS7142 ADC, address based on wiring
0x20, 0x21	PCA9535 I2C to GPIO, address based on wiring

Table 6.3: I2C addresses for relevant modules.

6.4 In Case of Upload Lock-up or Failure

If the Arduino IDE is having trouble uploading to the Arduino Nano RP2040 Connect, try double pressing (not too fast) the reset button on the Nano once the IDE starts trying to upload.

6.5 Arduino Programming Suggestions

In general, try to start from an existing set of code (an example for instance) when working on something. For example, if you need to use the APDS-9960, load one of the examples from the library, then modify it until it does what you want. Once you have written a few sketches, you might write a generic starting point for your subsequent sketches.

6.6 Arduino Button Setup

An example of how to setup the buttons on the CEC 325 board. Some important details to note is that the left button is attached to pin D9 (referred to as 9 in the Arduino infrastructure) which is attached to the RP2040 on the Arduino Nano RP2040 Connect module. The RP2040 has internal pull-up resistors that are tied correctly to the Arduino IDE so that the pins can be setup using `pinMode(LEFT_BUTTON_PIN, INPUT_PULLUP);` and do not require an external pull-up resistor. The right button is attached to pin A7 on the module which is routed to the WiFiNINA module. This module's internal pull-up is not implemented in the Arduino IDE (as of 2022 February 03). Therefore, it needs an external pull-up AND requires the sketch to include "WiFiNINA.h" AND the pin variable HAS to be declared using `#define` rather than a `const int`.

```
/* button_demo.ino
 *
 *  Gives an example of how to use the buttons on
 *  the CEC 325 board.
 *
 *  Seth McNeill
 *  2022 February 03
 */
```

```
#include "WiFiNINA.h"  // for A4-A7 and wifi/bluetooth

#define LEFT_BUTTON_PIN   9 // This input is on the
   RP2040 and has builtin pullup that works
#define RIGHT_BUTTON_PIN  A7 // This input is on the
   WiFiNINA and doesn't have working internal pullup

void setup() {
  Serial.begin(115200);
  //while(!Serial) delay(10);
  delay(2000);

  Serial.println("Starting...");

  pinMode(LEFT_BUTTON_PIN, INPUT_PULLUP);
  pinMode(RIGHT_BUTTON_PIN, INPUT);
  pinMode(LED_BUILTIN, OUTPUT);
}

void loop() {
  // note that the buttons read 1 (HIGH or true) when
   not pressed
  if(!digitalRead(LEFT_BUTTON_PIN)) {
    Serial.println("Left button pushed");
  }
  if(!digitalRead(RIGHT_BUTTON_PIN)) {
    Serial.println("Right button pushed");
  }
  delay(100); // keeps the loop from running too fast
   with nothing pushed
}
```

Listing 6.2: This is an example of how to setup the buttons on the CEC 325 board.

6.7 Arduino Serial Setup

The serial port has to be setup in the `setup()` function in the Arduino IDE. An example of setting it up is shown in Listing 6.3.

```
void setup() {
  Serial.begin(115200); // starts the serial
   connection at 115200 data (baud) rate
  // If you are attached to a computer (not a robot on
    battery power
  while(!Serial) delay(10); // wait for serial to
   start
  // delay(2000);  // if might be on battery, just
   wait a bit for it to start
  Serial.println("Starting...");

}

void loop() {
  Serial.println("This has a new line character at the
    end");
  Serial.print("This does not have a new line at the
   end: ");
  Serial.println(millis());
  delay(5000);
}
```

Listing 6.3: This is an example of how to start the serial port at 112,500 in the Arduino system.

I strongly suggest having the `setup()` function output to the serial port right after the serial port is started so that you can know that your board has booted. Note that if you do not have a serial port (your robot is running off of batteries so that it is not plugged into a computer) you need to remove the `while(!Serial) delay(10)` line and replace it with some sort of timeout function. A `delay(2000)` is usually sufficient to allow the serial to start if it exists but not hang if it doesn't. There are more complicated ways to do this, but that will be left as an exercise for the reader.

Chapter 7

Displays

7.1 Introduction

Adding a display to a device allows much more information to be shared from the device to the user than just using LEDs or buzzers. There are several types of displays that are commonly used in the embedded systems world.

7.1.1 LCD

The most common is a Liquid Crystal Display (LCD). Most of the computer monitors and computers are LCD. This technology gives good contrast, fast response (which gamers like), and minimal burn in. Old CRT displays had to have screensavers so that whatever was usually showing on the display wouldn't be there permanently. Thankfully modern displays don't typically have this problem. LCDs do require a backlight. This determines how bright the colors are. The pixels in the display just modulate how much of the backlight is showing.

At times you will find TFT displays. This stands for thin-film-transistor liquid-crystal display. They are a better version of LCD.

7.1.2 eInk

eInk displays are also available for embedded systems. These displays are of particular interest because they keep their display even when the power is turned off. This allows for very low power operations. The downsides are that they work off of reflected light so require special backlighting to be viewed at night and that they are very slow. A small display may take several seconds to refresh and some recommend not to update them more than once every few minutes if possible.

7.1.3 OLED

The cool part about Organic Light Emitting Diode (OLED) displays is that instead of each pixel blocking the backlight to make the display like in LCDs, in OLED displays each pixel is an LED that emits light. This makes OLED displays very bright with very good contrast ratios. They also have fast response times. Some OLED displays will get dimmer with time. Adafruit notes that for their small OLED displays the dimming becomes noticeable after about 1000 hours of being on. This is 41.7 days, so after a year of continuous use, an OLED display might not be very bright.

7.2 Pixel Layout

The layout of pixels on a display can be thought of as a cartesian coordinate system with a couple minor differences. First, pixels take up space, so the indexing is between the lines rather than on the lines. Second, the +Y axis points downward as shown in Figure 7.1. The units for the coordinates is always pixels and the coordinates are always integers.

Pixels can vary in size. This is important to keep in mind if you are trying to display something with a specific size. Pixel size varies from display to display so read carefully if you want something to display a specific size.

7.3 Using the Display

The display on the lab board is a 1.14” TFT display 240 pixels wide and 135 pixels high. Two libraries are required to use it:

1. `Adafruit_GFX.h` - this is the generic graphics library
2. `Adafruit_ST7789.h` - this is specific to our display

An example that shows much of the available functionality can be found (once the libraries are installed) at Examples → Adafruit ST7735 and ST7789

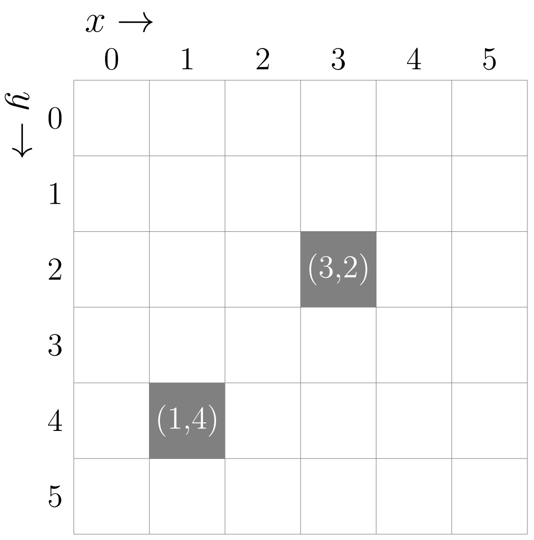

Figure 7.1: Pixels in a display occupy space are are referenced to the top left corner with the positive Y axis going down.

Library → graphicstest_st7789. The following have to be defined to use a display:

1. Width - how many pixels wide the display is
2. Height - how many pixels high the display is
3. Reset pin - Allows the microcontroller to reset the display
4. CS - Chip Select, used by SPI to select the display when communicating
5. DC - Data/Command pin used to determine the type of data being sent to the display

An example of creating a display instance is as follows:

```
Adafruit_ST7789 tft = Adafruit_ST7789(TFT_CS, TFT_DC, TFT_RST
);
```

Some useful methods that can be called on the display object are:

1. fillScreen - Fills the screen with the specified color. Usually used to clear the screen.
2. setTextSize - sets the size of the text (usually 2-4)

3. setTextColor - set what color the text should be. Some examples are
   (a) ST77XX_BLACK
   (b) ST77XX_BLUE
   (c) ST77XX_RED
   (d) ST77XX_YELLOW
4. print/println - these act the same as they do when using Serial
5. drawBitmap - draws a bitmap stored using PROGMEM or a canvas. It requires the following arugments
   (a) xpos - the x position for the image
   (b) ypos - the y position for the image
   (c) bitmap variable - the variables with the actual image
   (d) width - image width in pixels
   (e) height - image height in pixels
6. More can be found here.

7.3.1 Using Canvas to Reduce Flicker

Redrawing by calling `fillScreen()` causes a flicker that can be problematic. The solution is to create a canvas in memory that the program draws to and then copy it over to the display using `drawBitmap`. This technique does require that the microcontroller have enough memory to store the canvas.

The method requires creating a canvas variable, drawing everything to the canvas and then copying the canvas over to the display. Listing 7.1 shows the basics of this method.

```
// unlike the online example,
//this is a 16-bit color canvas
GFXcanvas16 canvas(240,135);
...
canvas.fillScreen(ST77XX_BLACK);  // clear canvas
canvas.setCursor(0,0);
canvas.setTextSize(3);
canvas.println("Hello world!");
```

```
    tft.drawRGBBitmap(0, 0, canvas.getBuffer(),
        240, 135);  // copy canvas to display
```

Listing 7.1: Snippets showing how to use a canvas to reduce flicker when updating a display.

Chapter 8

Sampling and Data Collection

8.1 Introduction

This chapter introduces students to the concepts of Analog-to-Digital Converters (ADCs) and data collection.

8.2 Sampling

The real world has signals that are continuous in both time and amplitude. Unfortunately, microcontrollers do not have continuous time or amplitude capabilities. This means that real signals have to be quantized in amplitude and sampled in time before they can be analyzed by a microcontroller. Figure 8.1 shows what quantization in amplitude looks like. The microcontroller can only see values of -3 to +3 in this example even though the real signal has values between these. Figure 8.2 includes the errors incurred due to quantization.

Sampling in the time domain could be done in a random fashion as demonstrated in Figure 8.3 but that is rarely a good idea. The math that can be done with a signal that is sampled at an even rate as shown in Figure 8.4 is very powerful. Therefore, we try very hard to sample at a specific frequency with minimal jitter in the time between samples.

A signal that has been both quantized and sampled is shown in Figure 8.5. The signal has been reduced to the sequence of numbers: {0,3,3,0,-3,-2}.

It is important to sample fast enough to actually capture the signal of interest. The minimum sampling rate to capture a particular signal is called

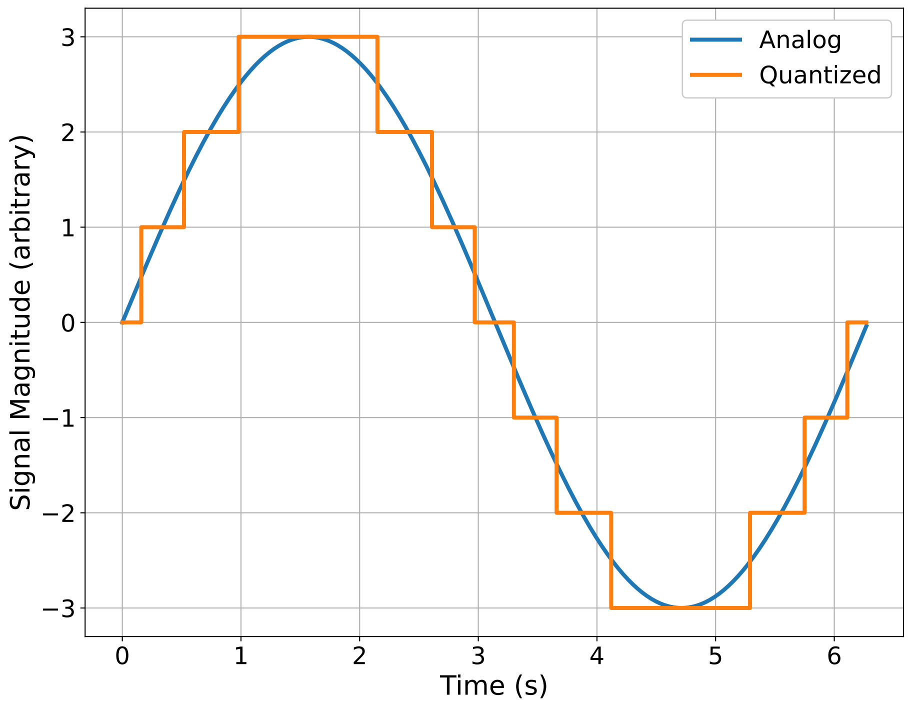

Figure 8.1: This shows amplitude quantization.

the Nyquist rate or Nyquist frequency. The sampling rate for a signal must be at least twice the highest frequency in the signal in order to correctly see the signal in a sampled system. Figure 8.6 shows the best case when sampling at exactly twice the frequency of the signal. This can be noted since there are two sample points in one period of the sine wave. The worst case of sampling at exactly the Nyquist rate is shown in Figure 8.7. A zoomed out version is shown in Figure 8.8. This is why it is best to give a little overhead when choosing the sample rate to use on a signal. If that is done, it yields Figure 8.9 where the sampling rate is just a little over the frequency of the sine wave.

In order to make sure that the signal never has too high frequencies, a circuit designer will typically have a low pass filter just before the input to the ADC.

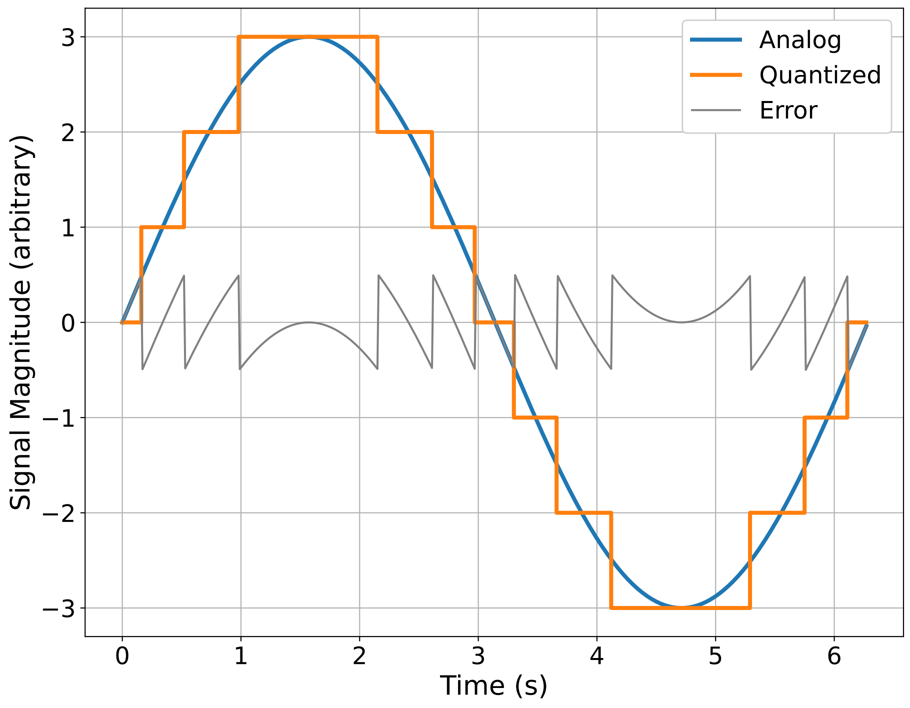

Figure 8.2: This shows amplitude quantization and the resulting error.

8.3 ADCs

Analog-to-digital converters (ADCs) are the piece of equipment that samples and quantizes a real signal. They have a set bit width. For instance a 10-bit ADC returns quantized values between 0 and 1023. Setting the sample rate can be done different ways depending on what the hardware capabilities are. Some ADCs take care of the sampling timing while others require the microcontroller to send a start conversion signal every time a new sample needs to be taken.

Once a sample is collected, it is often useful to convert it back into the voltage from the ADC count returned to the microcontroller. Equation 8.1 shows how this is done. The reference voltage is set somewhere in the system. It is a carefully constructed power supply to have as little noise as possible. In

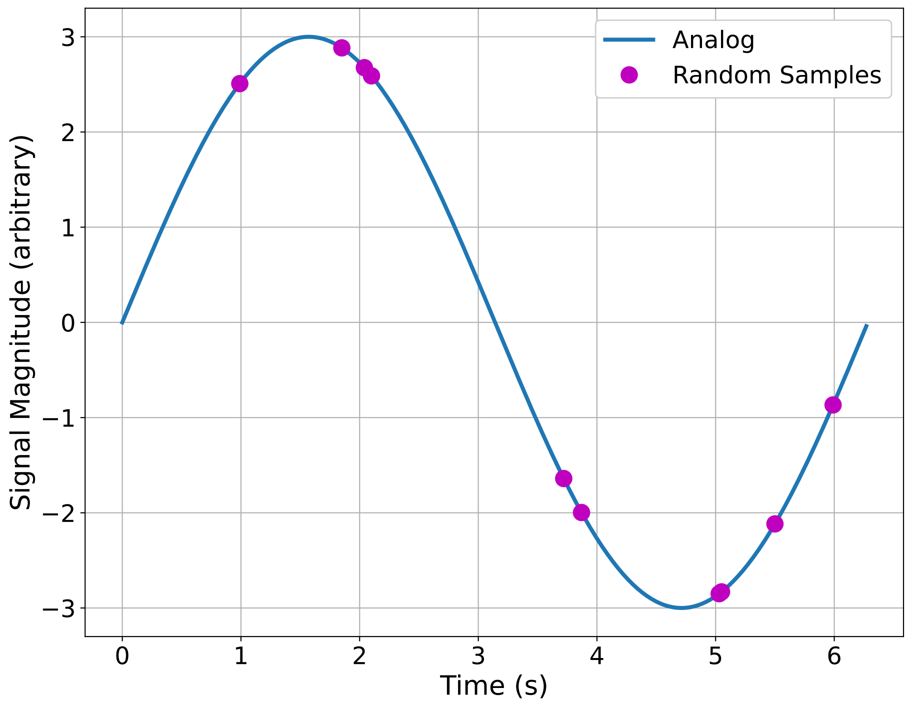

Figure 8.3: This shows random sampling.

the case of the Arduino Nano Connect RP2040, the onboard ADC reference is 3.3 V. The onboard ADCs are also 10-bit, so if an ADC reads 400 the applied voltage is 1.29 volts as shown in Equation 8.2. It can also be useful to know how many volts each count of the ADC is. This is shown in Equation 8.3 which gives 3.222 mV per count on the Nano's ADC.

$$\text{voltage} = \left(\frac{\text{ADCcount}}{\text{maxADCcount}}\right)\text{referenceVoltage} \tag{8.1}$$

$$\text{voltage} = \left(\frac{400}{1023}\right)3.3 = 1.29\text{V} \tag{8.2}$$

$$\text{volts/count} = \frac{referenceVoltage}{totalADCCounts} = \frac{3.3}{1024} = 3.222\text{mV/count} \tag{8.3}$$

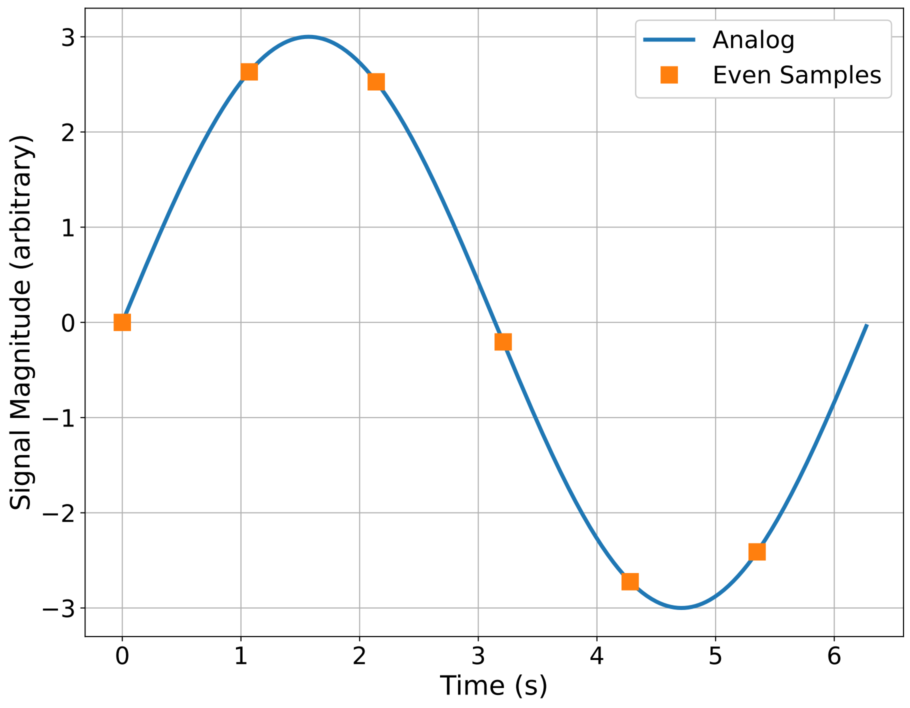

Figure 8.4: This shows even sampling.

8.4 Data Collection

8.4.1 Why Do We Collect Data?

One reason we collect data is to document something. It is important to record what has happened. One example of this is weather data that has been collected for centuries. This documentation then allows for analysis (another reason to collect data). Analysis of weather data over time led to

8.4.2 How Can We Tell If Data is Good?

8.4.3 Examples

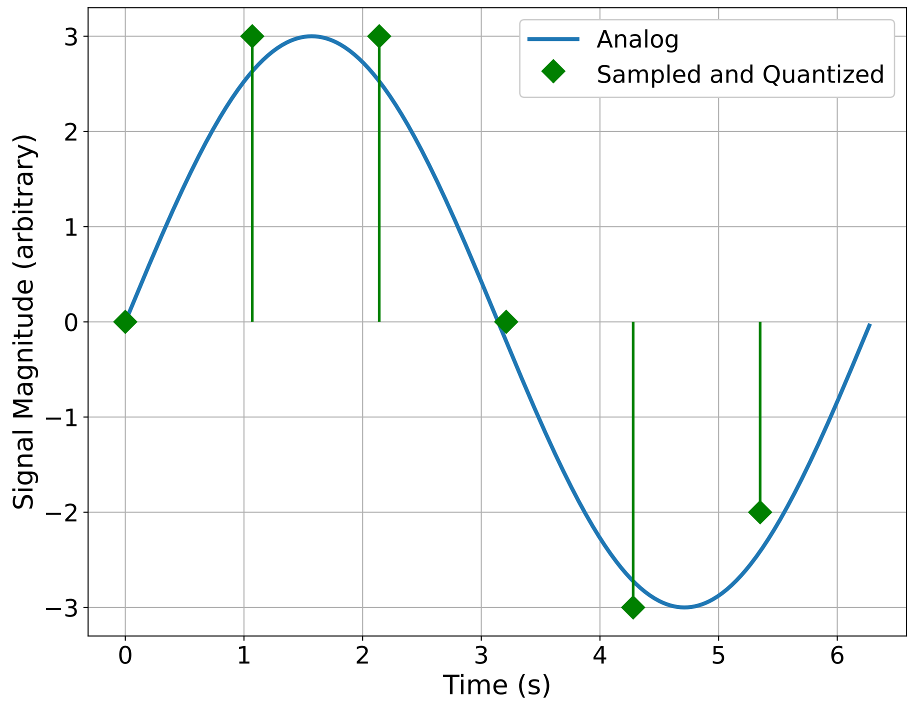

Figure 8.5: This shows a sampled and quantized signal.

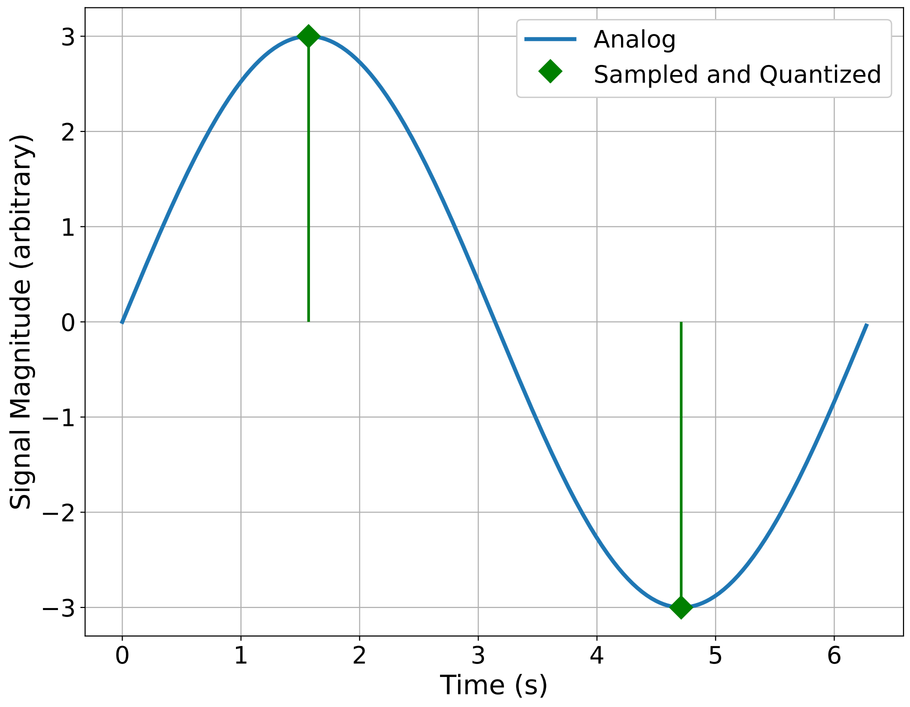

Figure 8.6: This is the best case when sampling at the Nyquist rate.

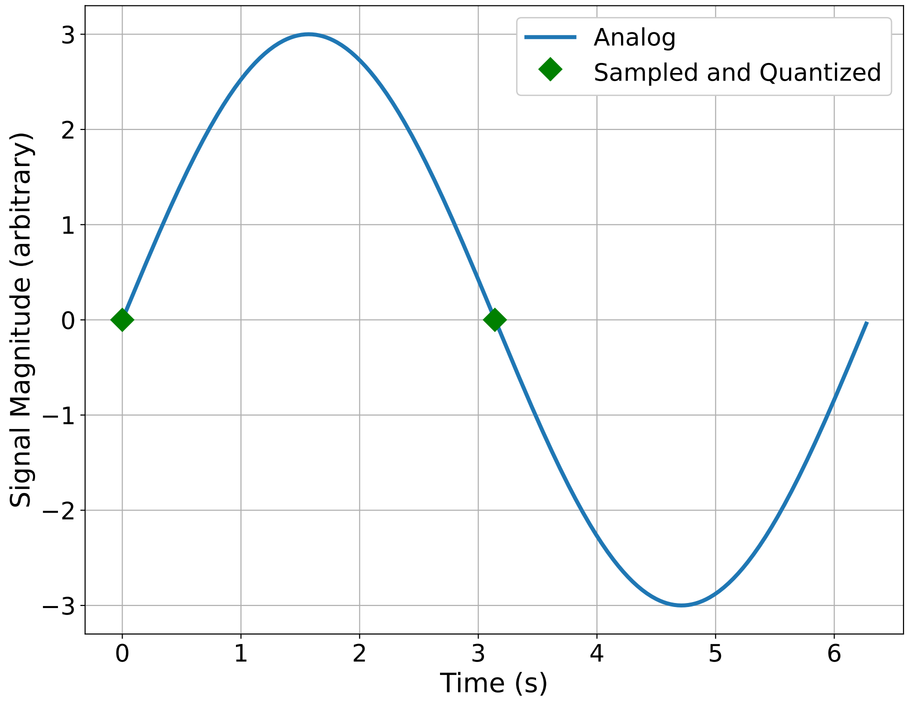

Figure 8.7: This is the worst case when sampling at exactly the Nyquist rate.

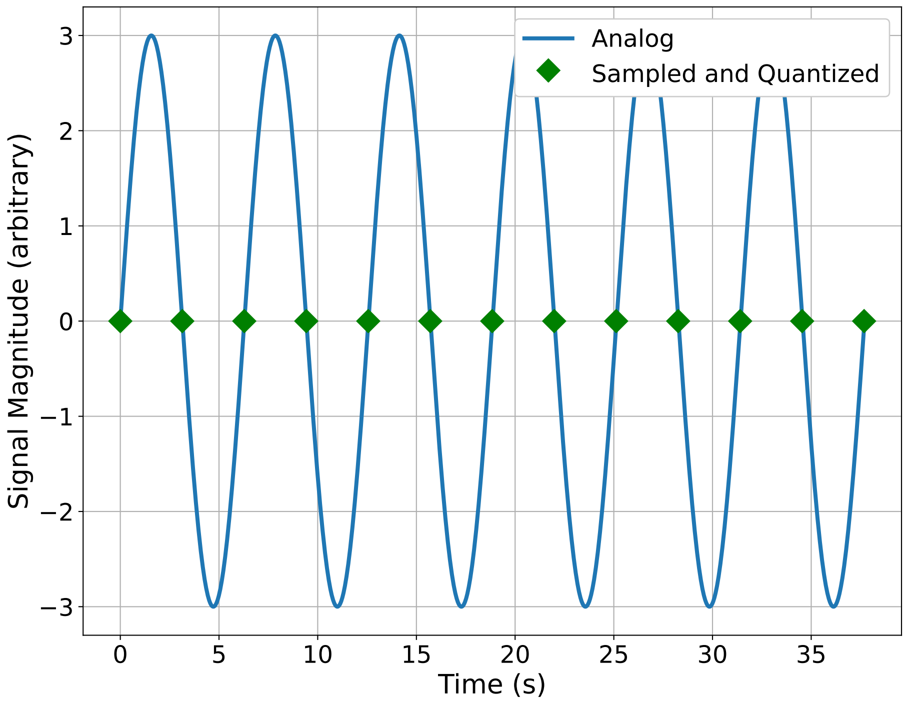

Figure 8.8: This is a zoomed out view of the worst case of sampling at exactly the Nyquist rate.

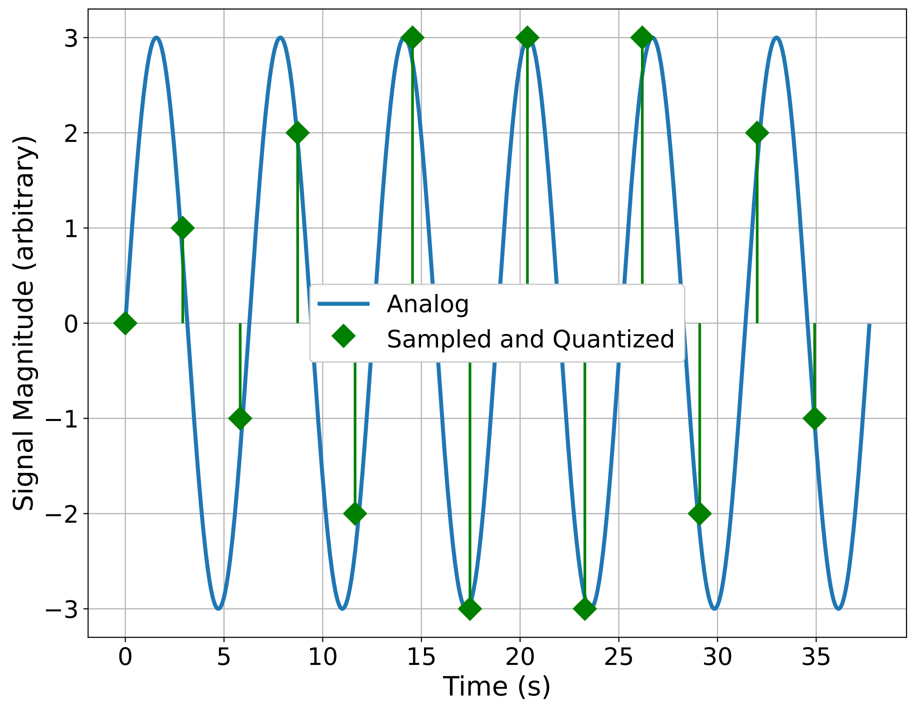

Figure 8.9: This is a zoomed out view of sampling just faster than the Nyquist rate.

Chapter 9

Inertial Measurements

9.1 Introduction

This chapter introduces students to using collecting inertial data such as linear and angular acceleration.

9.2 Rectilinear Kinematics

Rectilinear kinematics is about motion along a straight line. This provides a good starting point for discussing the mathematics of inertial measurements.

Time, position, velocity, and acceleration have the following differential relationships:

$$a = \frac{\mathrm{d}v}{\mathrm{d}t} \tag{9.1a}$$

$$v = \frac{\mathrm{d}s}{\mathrm{d}t} \tag{9.1b}$$

$$a \ \mathrm{d}s = v \ \mathrm{d}v \tag{9.1c}$$

If the acceleration is known (or can be assumed to be) constant, Equations 9.1 can be integrated to give Equations 9.2.

$$v = v_0 + a_c t \tag{9.2a}$$

$$s = s_0 + v_0 t + 0.5 a_c t^2 \tag{9.2b}$$

$$v^2 = v_0^2 + 2a_c(s - s_0) \tag{9.2c}$$

Constant acceleration is usually applied to the kinematics of projectiles where the constant acceleration is due to gravity. In the case of a digital IMU, the acceleration measurement is reported periodically (at 104 Hz in the default case of the LSM6DSOXTR on the Nano RP2040 Connect). Since it is a sampled system, we cannot just integrate Equations 9.1 to get velocity and position. What we do instead is to assume constant acceleration between samples and run a cumulative sum to calculate the velocity and position.

Integrating also assumes the knowledge of initial conditions. Usually we start with an initial condition of being at rest. This simplifies our starting point. At each subsequent calculation, the output of the previous sample is taken as the initial condition. This is shown in Equations 9.3.

$$v[k] = v[k-1] + a[k-1]\Delta t \tag{9.3a}$$

$$s[k] = s[k-1] + v[k-1]\Delta t + 0.5a[k-1](\Delta t)^2 \tag{9.3b}$$

As will become quite clear, this provides a noisy output with drift due to the two numeric integrations. There are several options for cleaning it up. One is to use trapezoidal rather than rectangular numeric integration. Another is to add some form of filtering to reduce the drift. Some options are a high pass filter on the acceleration and/or the velocity, Kalman filtering, or optimization techniques. A paper showing several of these techniques is Using Inertial Sensors for Position and Orientation Estimation.

9.3 Angle Measurement

Since earth's gravity provides a constant acceleration, detecting that constant allows a measurement of the angle between a device and it to determine the angle between. That is the basis for measuring the angle of a device using accelerometers. Since gyroscopes measure the rate of rotation, summing this angular speed over time will also give an estimate of angle. First, we will go over the basics of accelerometer angle measurement, then gyroscope angle measurement, and lastly, how to combine both measurements to correct the error each is prone to.

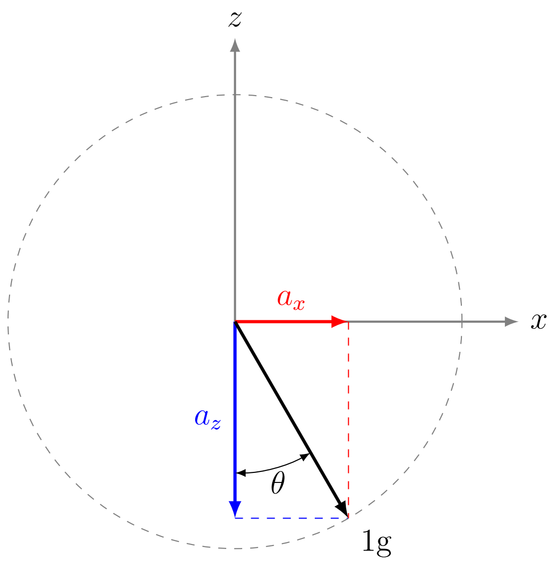

Figure 9.1: This shows the relative intensities of the accelerometer x and z (a_x and a_z respectively) measurements to the earth's gravity from the sensor frame of reference.

9.3.1 Accelerometer Angles

The problem is illustrated in Figure 9.1. In this case, we will take the x-axis as the horizontal axis and the z-axis is the vertical. We will assume that there are no external forces applied besides the force of gravity. This is faulty if the device is translating or otherwise not rotating about the IMU. However, we can still get some useful information even with this assumption. The vector sum of ax and az add to a point on the unit (1 g) circle. Therefore, the angle can be calculated as shown in Equation 9.4.

$$\theta_a = \tan^{-1}\left(\frac{a_x}{a_z}\right) = \operatorname{atan}2(a_x, a_z) \tag{9.4}$$

The atan2 function is provided in most languages. It is a “safe” version of $\tan^{-1}$ that deals with zeros and signs of a_x and a_z correctly and cleanly. If the language you are using has it (which it does) then use it to prevent calculation problems. Note that Equation 9.4 is not time dependant and does not have any numerical integration. It provides a ground truth of the angle based on the assumptions at the beginning: no forces other than gravity and no noise. Unfortunately, the output tends to have high frequency noise so you usually want to run the output through a low-pass filter.

9.3.2 Gyroscope Angles

Gyroscopes output angular velocity, ω, which when integrated can give the current angle relative to the starting angle. Since this system is sampled, instead of integration use a cumulative sum where each data point is based on the previous one. Equation 9.5 shows the mathematical approximation of the current angle based on the derivative of the angle θ with respect to time. This derivative is ω that is output by the gyroscope.

$$\theta(t + \Delta t) \approx \theta(t) + \frac{\partial}{\partial t}\theta(t)\Delta t \tag{9.5}$$

The actual implementation of Equation 9.5 is shown in Equation 9.6.

$$\theta_g[t] = \theta_g[t-1] + \omega\Delta t \tag{9.6}$$

Since this calculation is based on integration it is susceptible to drift. The error in the output will increase over time. A way to fix this is to run the output through a high pass filter to block the low frequency drift.

9.3.3 Fusing Accelerometer and Gyroscope Angles

The errors in accelerometer angle measurement are complementary to those in the gyroscope angle estimates since the acceleration estimates need a low pass filter and the gyroscope estimates need a high pass filter. If the values are combined carefully we can take advantage of the best properties of both measurements and use them to correct the errors in the other measurements. A complementary filter as shown in Equation 9.7 works well for this.

$$\theta_{mixed}[t] = \alpha\left(\theta_{mixed}[t-1] + \omega_{gyro}\Delta t\right) + (1-\alpha)\operatorname{atan2}(a_x, a_z) \tag{9.7}$$

As can be seen, if $\alpha = 0$, Equation 9.7 reduces to the accelerometer angle measurement and if $\alpha = 1$ it reduces to the gyroscope angle measurement. Choosing an α value between 0 and 1 gives a mixture of both results. Also, remember that the θ on the right side of Equation 9.7 is the previously calculated mixed θ, not the output from a separately calculated gyroscope angle as one might be tempted to do. Choose α through tuning (guided trial and error) based on the specific system that is being implemented and problem needing to be solved. In one example, I found a value of $\alpha = 0.95$ worked well.

It is very important to keep the units from each function correct. The gyroscopes on the Arduino Nano Connect RP2040 output in degrees per second. The inverse tangent functions typically return values in radians.

9.4 Useful References

1. Arduino constants https://forum.arduino.cc/t/pi-in-arduino/173649/3
2. https://stanford.edu/class/ee267/lectures/lecture9.pdf
3. https://stanford.edu/class/ee267/lectures/lecture10.pdf
4. https://stanford.edu/class/ee267/notes/ee267_notes_imu.pdf
5. ST Micro gravity subtraction

Chapter 10

Pulse Width Modulation

10.1 Introduction

This chapter introduces students to using Pulse Width Modulation (PWM) to control LED intensity and servo position.

10.2 References

1. PWM Tutorial (Circuit Digest)

Chapter 11

DC Motors and Control

11.1 Introduction

This chapter introduces students to types of DC motors and controlling DC motors using H-bridge type devices.

11.2 Types of DC Motors

The types of DC motors relevant to this class are as follows:

1. Brushed
   (a) DC
   (b) Hobby Servos
2. Brushless
3. Stepper

11.2.1 Brushed DC

Brushed DC motors are very common. The haptic motors that make your phone vibrate are likely brushed DC motors. The motors in most toys are also brushed DC motors. They are cheap to make and easy to use so they are very common. The direction of rotation is controlled by changing the polarity of the applied voltage. The speed of rotation is controlled by varying the magnitude of the applied voltage. The torque of a brushed DC motor increases with rotational speed. The advantages and disadvantages of brushed DC motors is outlined in Table 11.1.

Advantages	Disadvantages
Inexpensive	Mechanical noise from brushes
Lightweight	Electrical noise from brushes
Reasonably efficient	

Table 11.1: Advantages and disadvantages of brushed DC motors.

The brushes on brushed DC motors change which coil in the rotor is activated as the rotor rotates such that the rotor is always pushing away from the permanent magnets in the stator. The brushes transfer the current from the stator to the rotor by having electrical brushes contacting metal patches on the rotor.

11.2.2 Hobby Servos

11.2.3 Brushless DC Motors

Brushless DC motors have the coils in the stator and the permanent magnets on the rotor. The coils are activated in sequence to keep the rotor spinning. An electronic speed controller (ESC) controls the coil activation to keep the motor spinning at the desired rate. Some ESCs make use of a sensor on the motor to tell which coil needs activating to keep the motor spinning. Sensorless ESCs (common in the hobby market) measure the back emf (voltage across each coil) to know when to activate each coil.

It is important to choose a good quality ESC that is rated sufficiently to drive the motor. I have had a couple catastrophic failures of ESCs in flight. Fortunately, they were on fixed wing drones with good pilots so there was no other loss of payload/aircraft.

Brushless DC motors are used all around as well in things like computer fans, hard drive platter spinners, drones, and hybrid vehicles. The advantages and disadvantages of brushless DC motors are outlined in Table 11.2.

Advantages	Disadvantages
Quiet	Usually require separate ESC
Efficient	

Table 11.2: Advantages and disadvantages of brushless DC motors.

11.2.4 Stepper Motors

Stepper motors move one step at a time which makes them very useful in situations where fine motion control is needed. Position control is possible without any feedback mechanism when using stepper motors. However, if the motor is under too large a load, it may skip a step and position estimation will be off. Stepper motors have highest torque at low speed with torque dropping as speed increases. They require at least two H-bridges to drive.

11.3 References

1. Adafruit Motor Selection Guide
2. SparkFun Servo Tutorial
3. Arduino Servo Library
4. TI Stepper Motor Reference

Chapter 12

Deriving Information from Data

12.1 Introduction

This chapter introduces students to deriving useful information from raw data streams.

12.2 Collecting Good Data

Data collection is an important part of all of our lives. Everywhere we look, our eyes provide us with data. Our ears and nose also provide lots of data to our brains. However, it is important to realize why we might have a microcontroller collect data. Three reasons that come to mind are:

1. Documentation - to record for posterity of some kind (e.g. world speed records)
2. Actionable - do something based on the data (e.g. thermostat)
3. Analysis - better understanding of our world (e.g. climate research)

One big problem in life is to know whether the data we are collecting is good. Here are some ways to check data to determine if it is worth looking at:

1. Consistency

2. Comparison to ground truth
3. Repeatability
4. Multiple sensors
5. Common sense

12.2.1 Ground Truth

One very good method of checking data is to compare to something that is more accurate than what you are using. For instance, if you are using a thermometer that is accurate to ±2°C, compare it to a calibrated thermometer that is accurate to ±0.5°C. The calibration is important since it means that the more accurate thermometer has been tested against another thermometer that is even more accurate that itself. Most test equipment needs regular calibration and gets a tag of some kind on it when it does get calibrated. Check that tag before using it.

12.2.2 Repeatability

If it is possible to repeat a data collection multiple times, comparing results will give some idea of whether the data is good. The standard deviation is one statistic that can give a useful estimate of how repeatable an experiment is. If the standard deviation is high (defining "high" is important), than the data source probably is suspect.

12.2.3 Multiple Sensors (sources)

Another good way to check is to have multiple sources of the same information. If all three thermometers give the same result (within their error margins), then the data is more believable than just one thermometer. The shuttle ran with 5 computers with the general idea that there would be a vote between them in hopes that the majority would be correct. Sometimes it can be useful to have multiple sensors measuring the same thing but different ways. This can lead credence to the overall result.

12.2.4 Common Sense

Lastly, it is important to check all measurements with a great deal of common sense. If it doesn't feel cold to you and the thermometer says it is 0 degrees, then you should suspect the thermometer is wrong.

12.2.5 Data Collection Review

Check your data!

1. How much do you trust your source?
2. Does the data fit prior knowledge?
3. Does it make sense? (Sanity check)

12.3 Sample Timing

It is important to sample data at regular intervals. Jitter in sample timing can ruin the math of data analysis. The question is, how do we make sure that we are sampling at an even, regular interval? The simplest and most obvious method is shown in Listing 12.1. This method might work for data that is sampled at a very slow rate–less than once a second.

```
void loop()
{
    static uint32_t currTime;
    currTime = millis();

    if(currTime - lastSampleTime >= sampleInterval) {
        doSampling();
        lastToggle = currTime;
    }
}
```

Listing 12.1: This listing shows a simplistic way to time sampling.

However, that the reality is more like what is shown in Listing 12.2. There are often processes running during the loop that introduce jitter into the sampling period.

```
void loop()
{
    static uint32_t currTime;
    currTime = millis();

    if(currTime - lastSampleTime >= sampleInterval) {
        doSampling();
        SOMETHING_SLOW();
        lastToggle = currTime;
    }
    SOMETHING_ELSE_OCCASIONALLY_SLOW();
}
```

Listing 12.2: The problem with the simplistic approach is slow processes that introduce jitter.

So what is the solution? Interrupts!

12.3.1 Interrupts

Interrupts are as they sound: Normal code execution stops and the processor switches to a short piece of code called an Interrupt Service Routine (ISR or Interrupt Handler). Once the ISR finishes, code execution resumes where it had been interrupted. Interrupts can be triggered at specific repeatable times from onboard timers. They can also be triggered by other events. The thermometer on the board and the distance sensor can both be setup to trigger interrupts.

ISRs must be short. They cannot even have `Serial.println()` calls inside them. They typically do something like change a count, set a flag, toggle a value, start data collection (e.g. on an ADC), or read a value (e.g. also an ADC). If an ISR is too long regular code execution isn't resumed properly and the processor may reset and start over. This can be triggered by a watchdog timer.

12.3.2 Volatile Variables

It is important to designate any global variables that are changed by an ISR as `volatile`. This designation tells the compiler not to optimize the variable out. This would normally happen since the compiler does not see the ISR called anywhere in the code. An example of doing this is shown in Listing 12.3

```
volatile overtemp_flag = false;
void loop() {
    if(overtemp_flag) {
        overtemp_flag = false;
        // do something about being too warm
    }
}

interrupt void temp_isr() {
    overtemp_flag = true;
}
```

Listing 12.3: This code illustrates using volatile for variables used in ISRs.

12.4 Static Variables

The second useful variable declaration when using interrupts (but is also useful elsewhere) is `static`. The `static` declaration allows variables inside a function to persist between function calls, but keeps their scope inside the function. Listing 12.4 demonstrates this with the variable named `cnt` inside the function `count_int`. The first time `count_int` is called `cnt` is initialized to a value of 0. Then it is incremented to 1. The second time `count_int` is called `cnt` has a value of 1 even though without the `static` declaration it would have a value of 0 every time the function was called.

```
int count_int() {
    static int cnt = 0;
    return(cnt++);
}

void loop() {
    Serial.println(count_int());  // prints 0
    Serial.println(count_int());  // prints 1
}
```

Listing 12.4: This code demonstrates the functionality of a static variable.

12.5 Extrema Detection

I have found that I have often needed to find the peaks and valleys (the extrema) of a signal. When working on a computer Python (scipy.signal.find_peaks) and Matlab both have good algorithms for peak (valley) detection. When working on a microcontroller there are also libraries but it is important to understand what it is that you are looking for. Keep in mind that in an embedded (microcontroller) system we are usually processing data in real time and therefore need algorithms that work on live data coming in AND that do not take more processing power than is available between sample collections. Also, since the data is coming in continuously, a simple maximum measurement or minimum will not work since we do not have all the data.

Some data has a flat baseline with only positive peaks as shown in Figure 12.1 of humidity sensor data. The temperature data shown in Figure 12.2 shows a drifting baseline and also positive peaks. Figure 12.3 shows temperature and humidity together for the same time period.

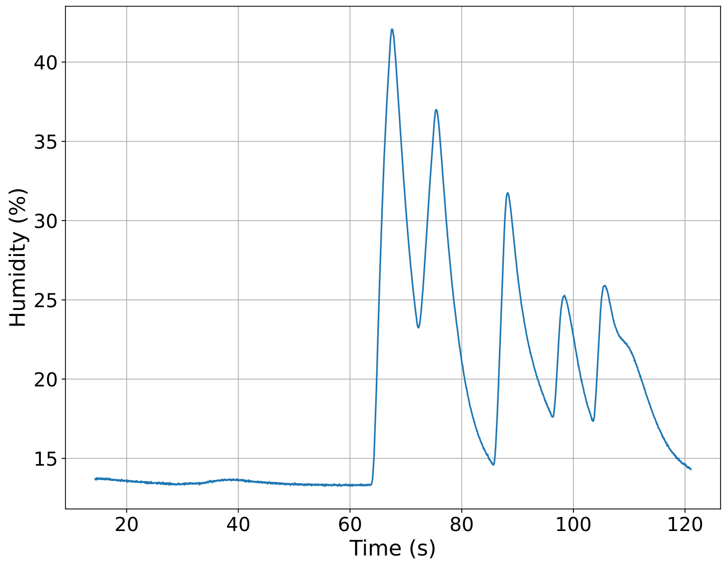

Figure 12.1: Humidity measurements like this (with a human blowing on the sensor) demonstrate positive peaks coming from a fairly stable baseline.

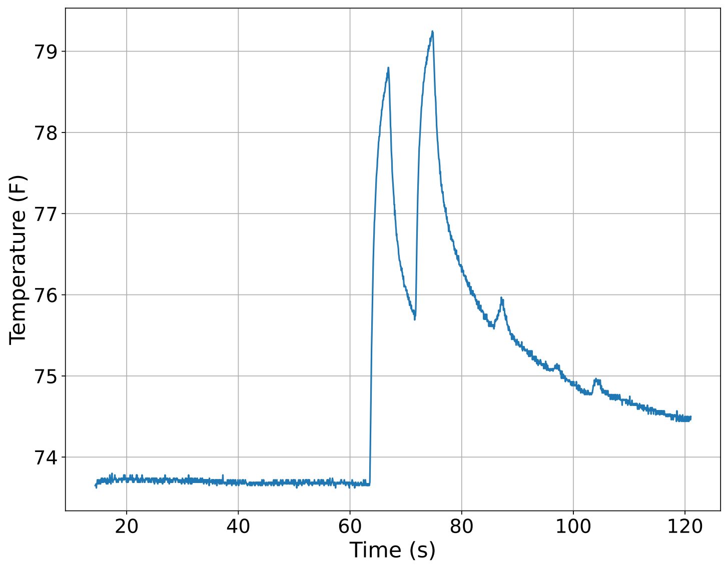

Figure 12.2: Temperature measurements like this (with a human blowing on the sensor) demonstrate positive peaks coming from a drifting baseline.

Other data has a baseline with positive and negative deviations such as the accelerometer data shown in Figure 12.4 Some algorithms are better for one or the other or may work for both situations.

The data so far has had pretty quiet baselines. Oftentimes, data can be much noisier as is illustrated in the light sensor data shown in Figure 12.5.

12.5.1 Thresholding

The simplest algorithm is a simple threshold system. This is used in most thermostats. It is measuring the temperature and if it gets above the threshold, it turns on the air conditioning. The threshold is now switched to a lower value (to prevent oscillations) and it now looks for the temperature to drop below the threshold. The first threshold required a positive derivative. The second threshold required a negative derivative. These are important distinctions.

Thresholding is simple to implement and can be useful in many situations.

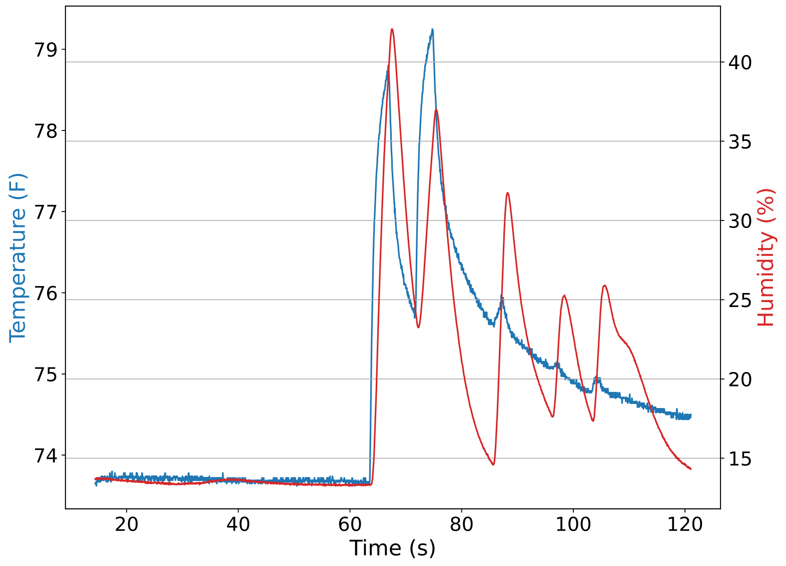

Figure 12.3: This shows both temperature and humidity on the same graph. Notice the different intensity of peaks between the two for the same events and teh difference in times for the peaks for the same events.

Obstacle detection is another example where we only care if the object is closer than some threshold. An example of a simple threshold valley detection is shown in Figure 12.6. Note that this method gives lots of points for each valley.

There is one more type of situation to think about. The potentiometer data shown in Figure 12.7 is an example where the baseline of the data shifts after an event.

12.5.2 Dispersion Method

To understand this version of extrema detection (and some others) a bit of terminology needs to be understood. The first is the baseline of a signal. This figure shows a signal that has a decreasing baseline. The baseline can be thought of as where the signal is when no extrema is present. It is often measured by some sort of low pass filter such as a moving average.

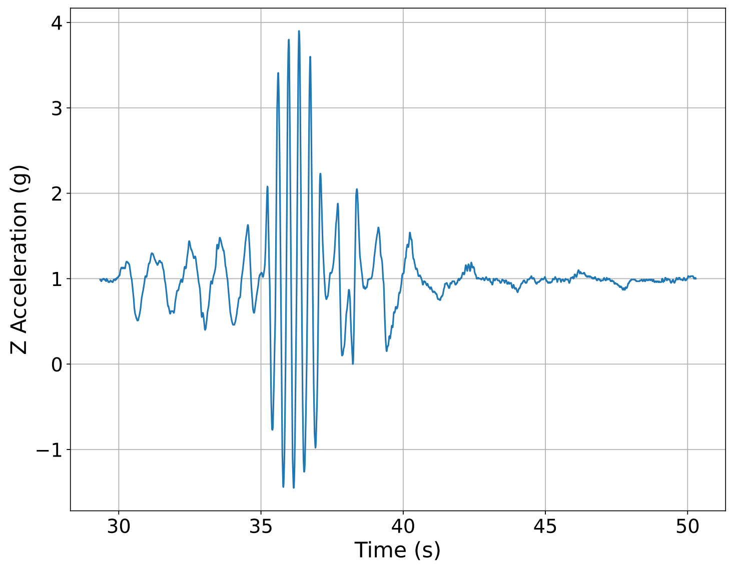

Figure 12.4: This accelerometer data demonstrates both positive and negative peaks from a baseline.

Once we determine the baseline of a signal, we can look for divergences from the baseline. One way to measure that divergence is to measure how many standard deviations away from the baseline a particular point lands. We can then set a threshold of how many standard deviations away from the baseline is considered an extrema.

For the dispersion/z-score method, the last parameter to look at is the influence which is a value between 0 and 1. An influence of 0 means that points that are considered extrema are not used in calculating the baseline. This does assume that the mean and standard deviation of the signal do not change over time. Since that is rarely, the case it is best to set the influence to something greater than 0. An influence of 1 means that extrema points influence the mean and standard deviation the same as all the other points. That is also rarely a good idea since averages are particularly sensitive to outliers and by definition, the extrema are outliers. A median filter or a simple exponential filter might work better for some situations.

As with all algorithms, the parameters will have to be chosen and tuned

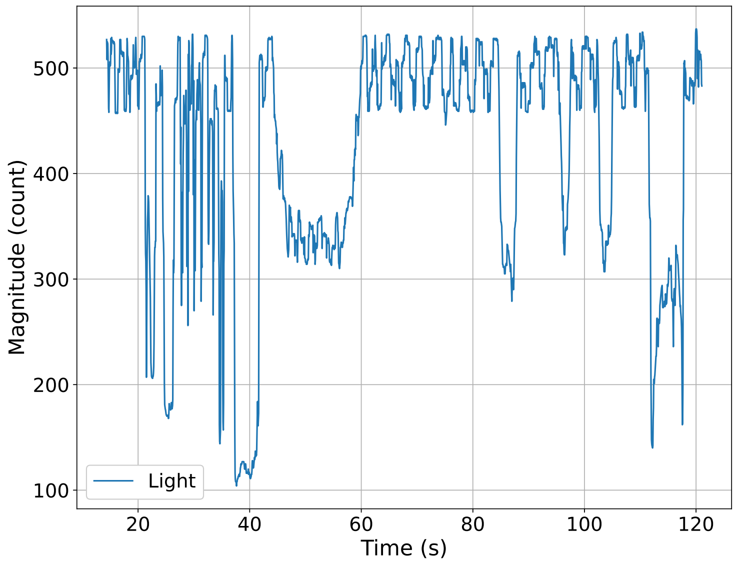

Figure 12.5: This light sensor (CDS cell into ADC) is noisy as sensor data can often be.

to work with the particular data you are measuring.

12.6 Bayes Theorem

A pattern classifier can be thought of as a set of functions, $g_i(\bar{x})$, called discriminant functions. There is a discriminant function for each class, ω_i, that evaluates a particular set of data, $\bar{x}$. The classification rule is then: Assign $\bar{x}$ to class ω_i if

$$g_i(\bar{x}) > g_j(\bar{x}) \text{ for all } j \neq i \tag{12.1}$$

A good and helpful option to explore for g_i is called Bayes Theorem. The formula for Bayes Theorem is shown in Equation 12.2.

$$P\left(\omega_i|\bar{x}\right) = \frac{p(\bar{x}|\omega_i)P(\omega_i)}{p(\bar{x})} \tag{12.2}$$

where

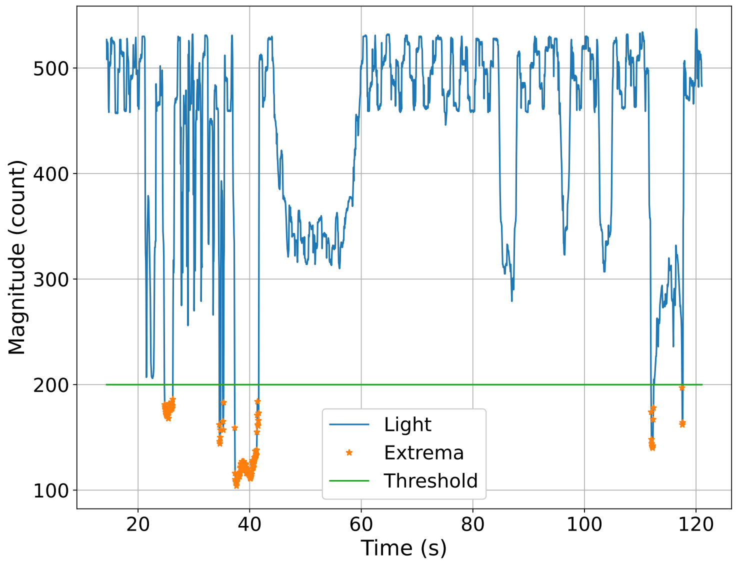

Figure 12.6: Looking for light values less than a threshold of 200 gives many points for each valley in this data.

- $\bar{x}$ is the data. If we are classifying faces is an image of a face
- ω_i is class i, one of the possible identities of the face we are trying to classify
- $P(\omega_i|\bar{x})$ is the posterior probability–the probability of ω_i given $\bar{x}$. In the example, it is the probability that the face image belongs to me rather than someone else in the database.
- $p(\bar{x}|\omega_i)$ is the likelihood or state conditional probability density function. This is the probability that $\bar{x}$ occurred given class ω_i. For example what is the probability of that the face image given that it is of me. The likelihood is calculated based on collected data.
- $P(\omega_i)$ is the prior probability of class ω_i. Rolling a single 6 sided die gives equal probability of each number so the priors are all 1/6. However, if instead we look at the sum of rolling 2 dice the probabilities of

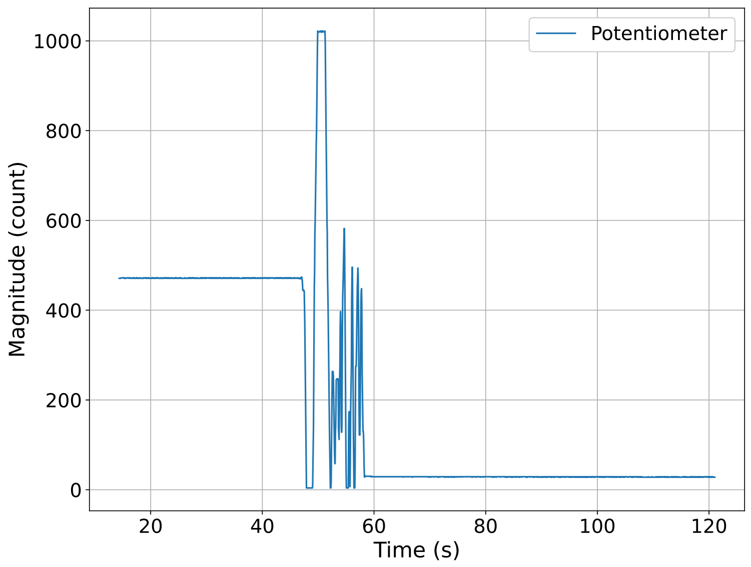

Figure 12.7: This potentiometer data shows the situation where the baseline shifts after an extrema event.

each number is no longer the same and the priors are different. The priors are calculated from existing, collected data.

- $p(\bar{x})$ is the evidence which is a scale factor to keep the probabilities summing to 1

$$\sum_j P(\omega_j|\bar{x}) = 1 \tag{12.3}$$

The evidence is calculated from existing, collected data as

$$p(\bar{x}) = \sum_j p(\bar{x}|\omega_j)P(\omega_j) \tag{12.4}$$

Discriminant functions are not unique. As long as $f(\cdot)$ is monotonically increasing $f(g_i(\bar{x}))$ will work too. Therefore, all of the Equations 12.5 will work.

$$
\begin{aligned}
g_i(\hat{x}) =& P(\omega_i|\bar{x}) && (12.5a)\\
=& \frac{p(\hat{x}|\omega_i)P(\omega_i)}{\sum_j p(\hat{x}|\omega_j)P(\omega_j)} && (12.5b)\\
=& p(\hat{x}|\omega_i)P(\omega_i) && (12.5c)\\
=& \ln\left(p\left(\hat{x}|\omega_i\right)\right) + \ln\left(P\left(\omega_i\right)\right) && (12.5d)
\end{aligned}
$$

12.7 ML Metrics

12.7.1 Accuracy

$$\text{Accuracy} = \frac{\#\text{correct}}{\#\text{predictions}} \tag{12.6}$$

12.7.2 Precision

$$\text{Precision} = \frac{\#\text{TruePositives}}{\#\text{TruePositives} + \#\text{FalsePositives}} \tag{12.7}$$

12.7.3 Recall

$$\text{Recall} = \frac{\#\text{TruePositives}}{\#\text{TruePositives} + \#\text{FalseNegatives}} \tag{12.8}$$

12.7.4 F1 Score

$$\text{F1Score} = 2 * \frac{\text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}} \tag{12.9}$$

12.7.5 References

1. This is a very good discussion of real-time peak detection algorithms and examples
2. https://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-10-4
3. Very simple peak detector that just looks for points above a threshold
4. An Arduino library for dispersion method

Chapter 13

Control of Systems

13.1 Introduction

This chapter introduces students to some basic control strategies including PID.

13.2 PID Control

The parallel (how we usually draw the controller) form of the PID equation is shown in Equation 13.1.

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{d}{dt} e(t) \tag{13.1}$$

The standard form of the PID equation rearranges the gains so that they have a more easily understood physical meaning as shown in Equation 13.2.

$$u(t) = K_p \left(e(t) + \frac{1}{T_i} \int_0^t e(\tau) d\tau + T_d \frac{d}{dt} e(t) \right) \tag{13.2}$$

In the standard form, T_i is the time it will take to eliminate all errors assuming the loop control does not change. T_d is how far into the future the derivative term is trying to predict the error. Note that times in this context can either be in seconds or samples. Samples is more typical in an actual implementation.

13.2.1 Proportional Control

Sometimes just using the proportional term is enough for controlling a system. The equation is shown in Equation 13.3.

$$u(t) = K_p e(t) \tag{13.3}$$

13.2.2 Integral Control

It is a rare system that only requires integral control but the equation for integral control is shown in Equation 13.4.

$$u(t) = K_i \int_0^t e(\tau) d\tau \tag{13.4}$$

13.2.3 Differential Control

It is also rare that a system can be controlled satisfactorily with only differential control, but for completeness it is shown in Equation 13.5.

$$u(t) = K_d \frac{d}{dt} e(t) \tag{13.5}$$

13.2.4 PI and PD Control

PI and PD control are sometimes sufficient to control a system satisfactorily.

Chapter 14

WiFI and Bluetooth

14.1 Introduction

This chapter introduces students to Wifi and Bluetooth.

14.2 Updating WiFiNINA Firmware

For the 1.8 versions of the Arduino IDE, the WiFiNINA firmware is bundled with the IDE. This means that the IDE needs to be upgraded to get the latest version of the firmware.

Once you have the most recent version of the IDE, follow the directions here to upgrade the firmware.