IoT Bootcamp Instructor Manual

Brian Rashap

March 5, 2020

Contents

List of Figures

List of Tables

List of Code Listings

Preface

We are in the midst of a revolution where smart, connected devices will change all aspects of live. Dubbed the Internet of Things [1] (IoT), we are seeing the proliferation of IoT devices into all facets of society. We are entering the age of advanced manufacturing, what is being referred to as Industry 4.0, where predictive analytics, generative design, advanced materials, ubiquitous sensors, and automation/robotics is revolutionizing industries. Where we live, what started out as smart thermostat has become a connected home complete with robotic vacuums, connected appliances, smart security systems, and voice assistants. Our daily routines outside the home will also be revolutionized as the connected city helps us navigate traffic, find parking, optimize public transportation, improve air quality, reduce noise, and enhance public safety. From wearables to autonomous automobiles, technology is impacting every aspect of our lives. Behind all of this is the Internet of Things, low cost yet powerful compute capability, wirelessly connected to each other and the Cloud, and able to sense/influence the environments where we work and live. It is estimated that there will be 1 trillion connected devices by 2035. That is more than 100 smart devices for every person on earth.

0.1 The First Three Industrial Revolutions

Greek mythology tells the story of Prometheus, a Titan and the ultimate trickster, steals fire from the gods and in giving it to humanity starts civiliza-

[1] The term was first used by Kevin Ashton of the Auto-ID Center at MIT in 1999

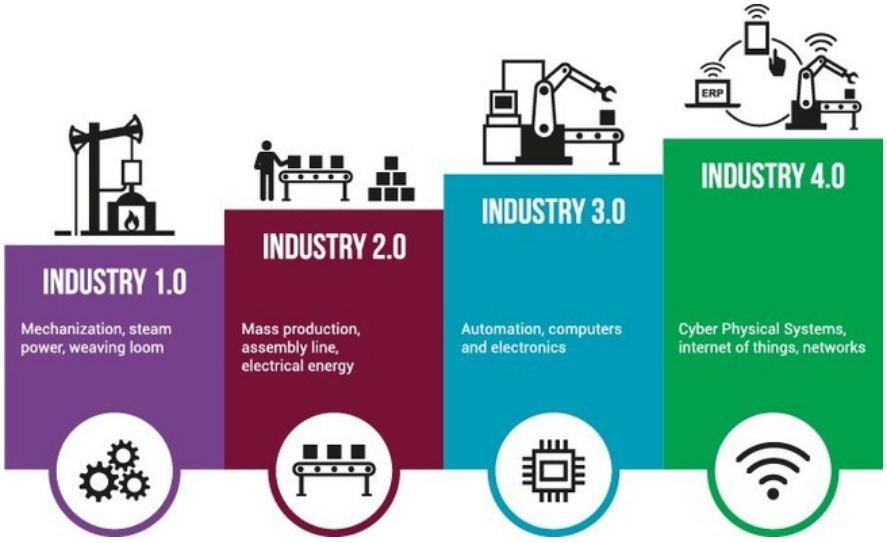


Figure 1: Evolution of Industry

tion. Since then, mankind has harnessed its innate creativity to continually evolve technology. While innovation often happens at a steady pace, there are been four revolutions that have changed the industry and thus civilization as a whole.

The First Industrial Revolution (late 18^{th} and early 19^{th} centuries) saw the emergency of mechanization powered by the invention of the steam engine and the leaving loom. The world got "smaller" due to new types of transportation: the railroads and steam-powered ships. The first factories emerged and the modern city was born.

In the Second Industrial Revolution (late 19^{th} century), new technology advancements led to the emergence of new energy resources (electricity, gas, and oil), an exponential demand for steel, and harnessing chemical synthesis. Communication was revolutionized by he invention of the telegraph and telephone. At the turn of the century, the world once again got "smaller" with the rise of the automobile and the airplane. Probably the most impactful and defining aspect of this industrial revolution was the centralization of research, the emergence of large factories, and the organizational struc-

tures for production developed by Frederick Winslow Taylor [2] and actualized (independently) by Henry Ford at the Ford Motor Company [3]

In the second half of the 20^{th} century, the Third Industrial Revolution (and the Information Age) was powered by the invention of the transistor. This led to the rise of electronics, the invention of the microprocessor, and a revolution in communications. Miniaturization opened the door on innovations in space and bio-technologies. The personal computer and the early internet democratized access to information.

0.2 Industry 4.0

Industry 4.0 has both expanded the possibilities of digital transformation and increased its importance to the organization. Industry 4.0 combines and connects digital and physical technologies—artificial intelligence, the Internet of Things, additive manufacturing, robotics, cloud computing, and others—to drive more flexible, responsive, and interconnected enterprises capable of making more informed decisions.

This Fourth Industrial Revolution carries with it seemingly limitless opportunity—and seemingly limitless options for technology investments. As organizations seek digital transformation, they should consider multiple questions to help narrow their choices: what, precisely, they hope to transform; where to invest their resources; and which advanced technologies can best serve their strategic needs. Further, digital transformation cannot happen in a vacuum; it does not end simply with implementing new technologies and letting them run. Rather, true digital transformation typically has profound implications for an organization—affecting strategy, talent, business models, and even the way the company is organized.

This revolution could be the first to deviate from the energy-greed trend—in terms of nonrenewable resources—because we have been integrating more and more possibilities to power our production processes with alternative resources. Tomorrow, Industry 4.0 factories will be embedded in smart cities

[2] Principles of Scientific Management, 1911.

[3] Highly specialized machinery, the assembly line, and the $5 a day wage.

and powered by wind, sun and geothermal energy.

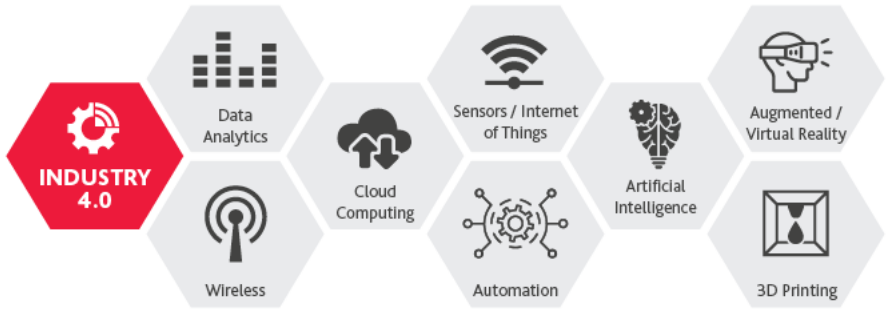


Figure 2: Components of Industry 4.0

0.3 Internet of Things

The Internet of Things is an emerging topic of technical, social, and economic significance. Consumer products, durable goods, cars and trucks, industrial and utility components, sensors, and other everyday objects are being combined with Internet connectivity and powerful data analytic capabilities that promise to transform the way we work, live, and play. Projections for the impact of IoT on the Internet and economy are impressive, with some anticipating as many as 100 billion connected IoT devices and a global economic impact of more than $11 trillion by 2025.

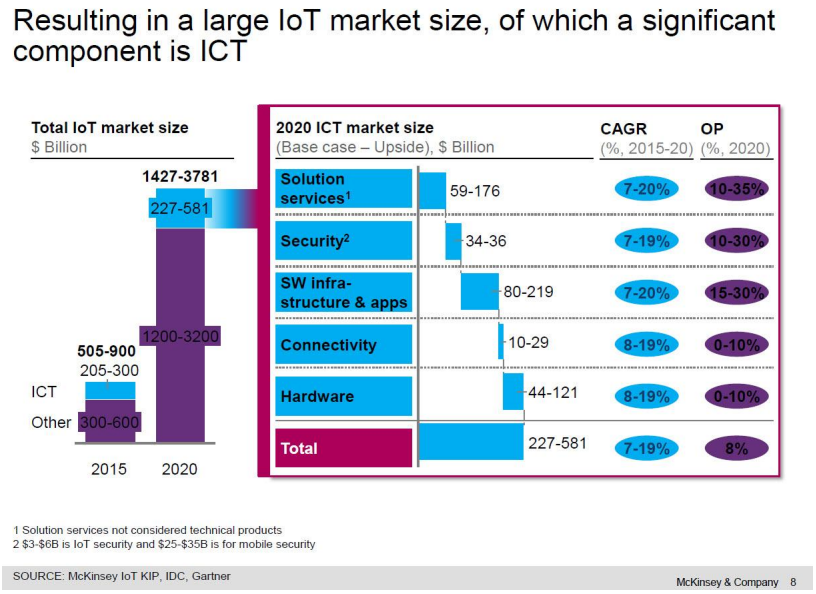


Figure 3: IoT Market Size

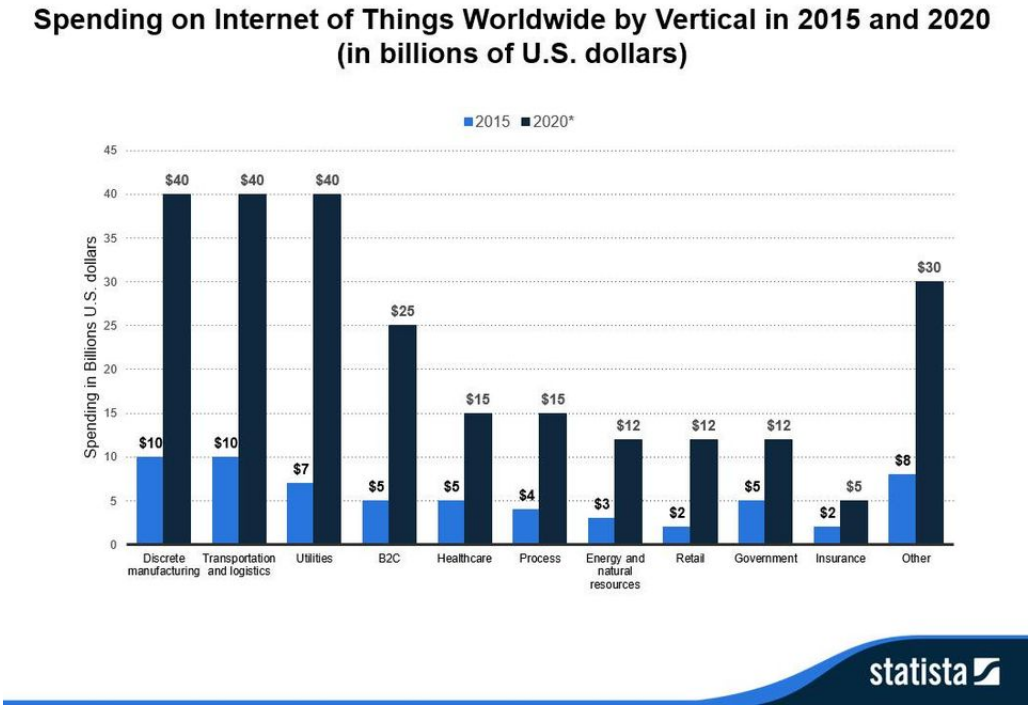


Figure 4: Spending by IoT Vertical

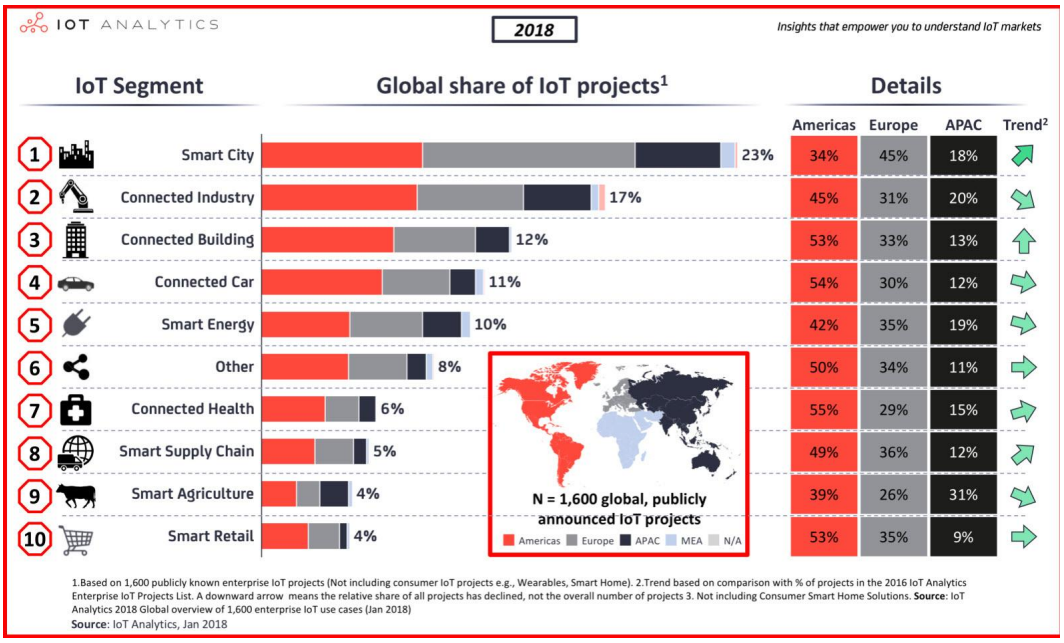


Figure 5: IoT Segments

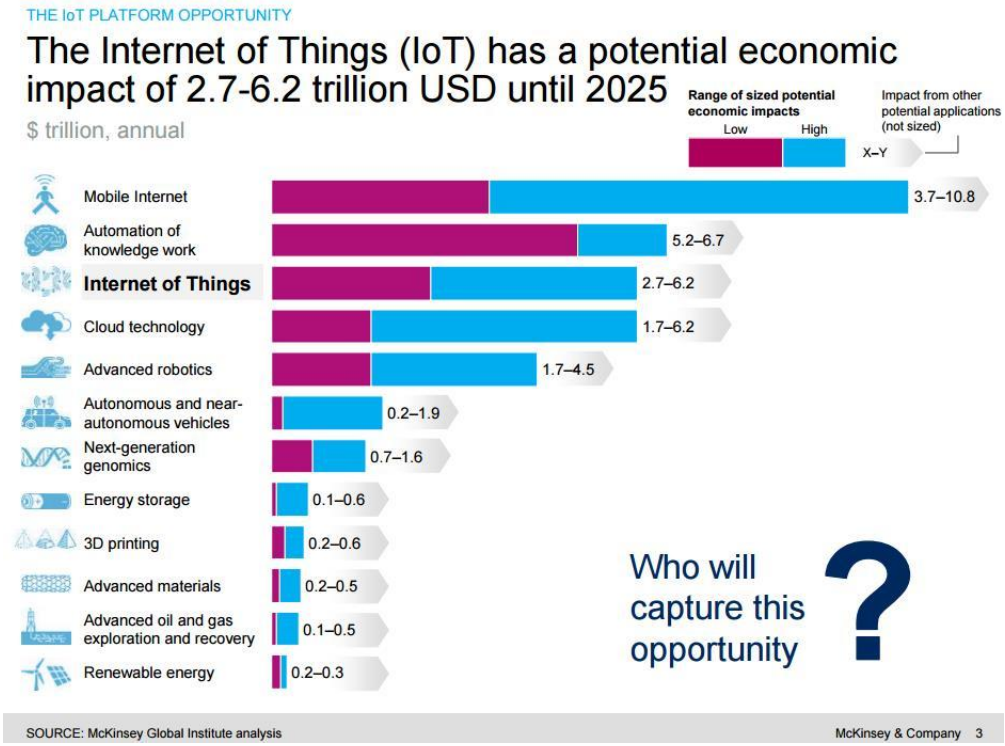


Figure 6: IoT Potentail Economic Impact

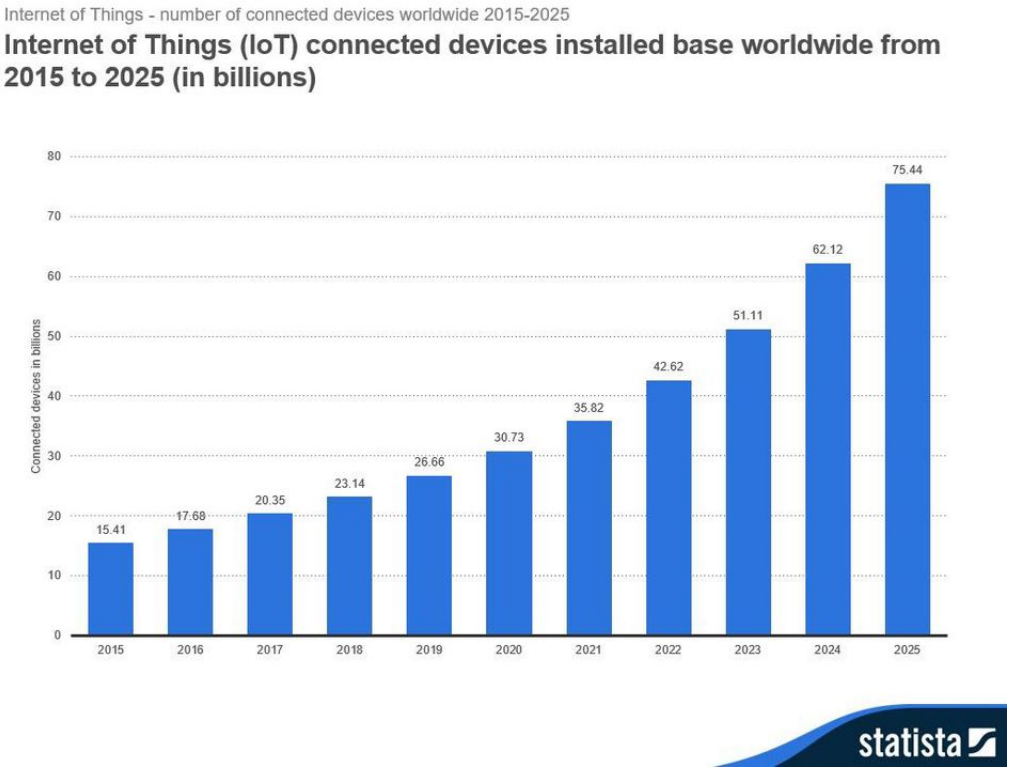


Figure 7: IoT Growth

Chapter 1

Basic Electronics

1.1 Basic Units

Throughout this course, we will be referring to values that are very small or very large. We will be using the scientific notation outlined in the table 1.1.

Submultiple	Prefix	Symbol	Multiple	Prefix	Symbol
10^{-1}	deci	d	10	deca	da
10^{-2}	centi	c	10^{2}	hecto	h
10^{-3}	milli	m	10^{3}	kilo	k
10^{-6}	micro	μ	10^{6}	mega	M
10^{-9}	nano	n	10^{9}	giga	G
10^{-12}	pico	p	10^{12}	tera	T
10^{-15}	femto	f	10^{15}	peta	P
10^{-18}	atto	a	10^{18}	exa	E

Table 1.1: Scientific Notation

1.2 Resistors, Capacitors, Inductors

Electricity is the movement of charged particles, electrons. There a four principles of electricity that will be important to our exploration of the Internet of Things.

- Voltage (V) is the difference in charge between two points.
- Current(I) is the rate at which charge is flowing.
- Resistance (R) is a material's tendency to resist the flow of charge (current).
- Power (P) is the transfer of energy over time.

These principles describe the movement of charge, and thus the behavior of electrons. Thoughout this course, we will explore various circuits (a closed loop system that allows charge to move from one component to another). By controlling the movement of the charge, the circuit allows us to do work.

Ohm's Law

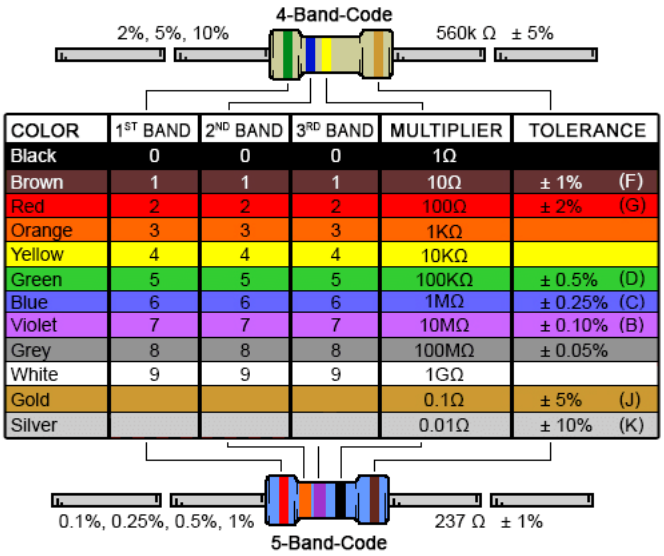
One of the first principles that we need to understand circuits is Ohm's law. Georg Ohm discovered that the amount of electric current through a metal conductor in a circuit is directly proportional to the voltage impressed across it. Ohm expressed his discovery in the form of a simple equation (Equation 1.1) that described how voltage, current, and resistance are interrelated.

$$V = I * R \tag{1.1}$$

1.2.1 Resistor Basics

Resistors are electronic components which have a specific resistance. The resistor's resistance limits the flow of electrons through a circuit. The elec-

trical resistance of a resistor is measured in ohms. The symbol for an ohm is the greek capital-omega: Ω. The (somewhat roundabout) definition of 1Ω is the resistance between two points where 1 volt (1V) of applied potential energy will push 1 ampere (1A) of current. The resistance is noted by the color bars on the resistor in accordance to format see in Figure **??**.



COLOR	1ST BAND	2ND BAND	3RD BAND	MULTIPLIER	TOLERANCE
Black	0	0	0	1Ω	
Brown	1	1	1	10Ω	± 1% (F)
Red	2	2	2	100Ω	± 2% (G)
Orange	3	3	3	1KΩ	
Yellow	4	4	4	10KΩ	
Green	5	5	5	100KΩ	± 0.5% (D)
Blue	6	6	6	1MΩ	± 0.25% (C)
Violet	7	7	7	10MΩ	± 0.10% (B)
Grey	8	8	8	100MΩ	± 0.05%
White	9	9	9	1GΩ	
Gold				0.1Ω	± 5% (J)
Silver				0.01Ω	± 10% (K)

Figure 1.1: Resistor Color Bands

They are passive components, meaning they only consume power (and can't generate it). Resistors are usually added to circuits where they complement active components like op-amps, microcontrollers, and other integrated circuits. Commonly resistors are used to limit current, divide voltages, and pull-up I/O lines. We will discuss these applications later.

1.2.2 Capacitor Basics

A second type of passive component that is common in electrical circuits is the capacitor. Capacitors have the ability to store electrical charge, so they are in a way like a fully charged battery. Common applications for a capacitor include local energy storage, voltage spike suppression, and complex signal

filtering. A diagram of an idealized capacitor can be seen in Figure **??**.

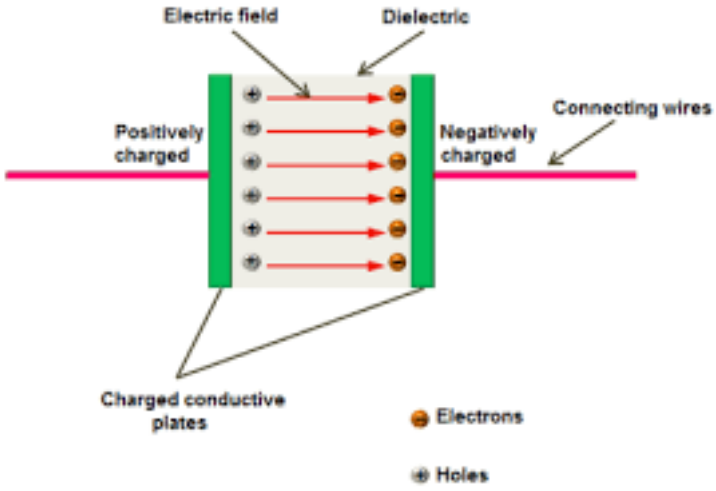


Figure 1.2: Capacitor in a simple circuit

Capacitor is characterized by a constant capacitance C, in farads (F), defined as the ratio of the positive or negative charge Q on each conductor to the voltage V between them (Equation 1.2).

$$C = \frac{Q}{V} \tag{1.2}$$

A capacitance of one farad means that one coulomb[1] of charge on each conductor causes a voltage of one volt across the device.

1.2.3 Inductor Basics

The third type of passive component is an inductor. An inductor typically consists of an insulated wire wound into a coil around a core. An inductor stores energy in a magnet field when electric current is flowing through it.

[1] 1 Coulomb = $6.24 * 10^18$ charged particles

When the current flowing through an inductor changes, the time-varying magnetic field induces an electromotive force (e.m.f.) (voltage) in the conductor, described by Faraday's law [2]. Lenz law [3] states that the direction of the current induced in a conductor by a changing magnetic field is such that the magnetic field created by the induced current opposes the initial changing magnetic field. As a result, inductors oppose any changes in current through them.

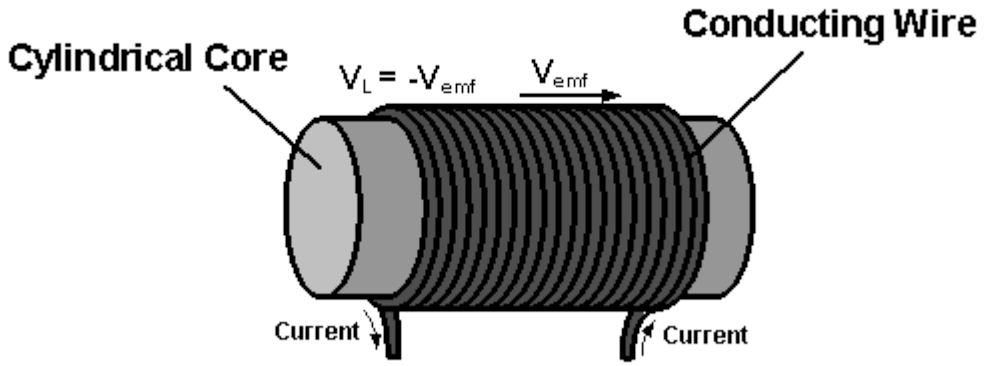


Figure 1.3: Inductor diagram

An inductor is characterized by its inductance, which is the ratio of the voltage to the rate of change of current. The unit of inductance is the henry (H). In the measurement of magnetic circuits, it is equivalent to weber/ampere. Inductors have values that typically range from 1 μH to 20 H. Many inductors have a magnetic core made of iron or ferrite inside the coil, which serves to increase the magnetic field and thus the inductance. Inductors are widely used in alternating current (AC) electronic equipment, particularly in radio equipment. They are used to block AC while allowing DC to pass; inductors designed for this purpose are called chokes. They are also used in electronic filters to separate signals of different frequencies, and in combination with capacitors to make tuned circuits, used to tune radio and TV receivers.

[2]Faraday's law states that the absolute value or magnitude of the circulation of the electric field E around a closed loop is equal to the rate of change of the magnetic flux through the area enclosed by the loop.

[3]ADD A FOOTNOTE

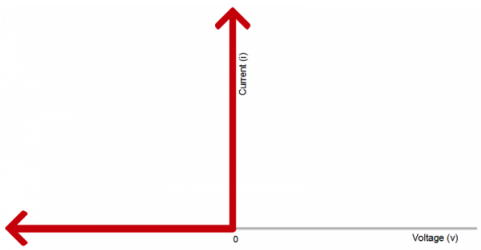
1.3 Diodes

The key function of an diode is to control the direction of current-flow. Current passing through a diode can only go in one direction, called the forward direction. Current trying to flow the reverse direction is blocked. They're like the one-way valve of electronics.

If the voltage across a diode is negative, no current can flow, and the ideal diode looks like an open circuit. In such a situation, the diode is said to be off or reverse biased.

As long as the voltage across the diode isn't negative, it'll "turn on" and conduct current. Ideally a diode would act like a short circuit (0V across it) if it was conducting current. When a diode is conducting current it's forward biased (electronics jargon for "on"). An ideal diode's response to voltage bias can be seen in Figure 1.4.

Figure 1.4: Ideal Diode Response



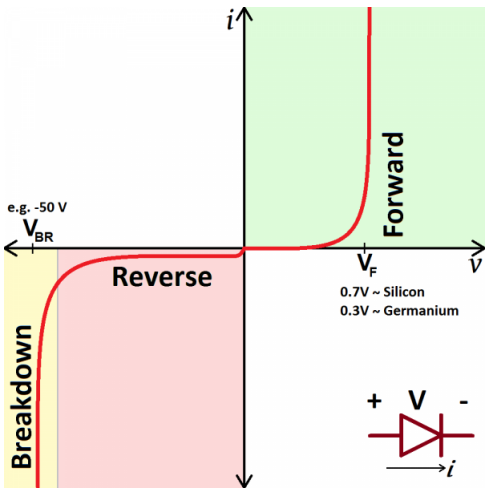
1.3.1 Real Diodes

However, there is no such thing as an ideal diode. Diodes do consume some amount of power when conducting forward current, and they won't block out all reverse current. Real-world diodes are a bit more complicated, and they all have unique characteristics which define how they actually operate.

The most important diode characteristic is its current-voltage (i-v) relationship. This defines what the current running through a component is, given

what voltage is measured across it. Resistors, for example, have a simple, linear i-v relationship...Ohm's Law. The i-v curve of a diode, though, is entirely non-linear, as seen in Figure 1.5.

Figure 1.5: Actual Diode Response



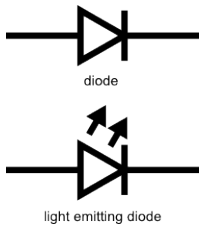
Depending on the voltage applied across it, a diode will operate in one of three regions:

- Forward bias: When the voltage across the diode is positive the diode is "on" and current can run through. The voltage should be greater than the forward voltage (VF) in order for the current to be anything significant.
- Reverse bias: This is the "off" mode of the diode, where the voltage is less than VF but greater than -VBR. In this mode current flow is (mostly) blocked, and the diode is off. A very small amount of current (on the order of nA) – called reverse saturation current – is able to flow in reverse through the diode.
- Breakdown: When the voltage applied across the diode is very large and negative, lots of current will be able to flow in the reverse direction, from cathode to anode.

1.3.2 Light Emitting Diodes (LEDs)

LEDs (that's "ell-ee-dees") are a particular type of diode that convert electrical energy into light. In fact, LED stands for "Light Emitting Diode." And this is reflected in the similarity between the diode and LED schematic symbols as seen in Figure 1.6.

Figure 1.6: Light Emitting Diode Symbol



In short, LEDs are like tiny lightbulbs. However, LEDs require a lot less power to light up by comparison. They're also more energy efficient, so they don't tend to get hot like conventional lightbulbs do (unless you're really pumping power into them). This makes them ideal for mobile devices and other low-power applications. Don't count them out of the high-power game, though. High-intensity LEDs have found their way into accent lighting, spotlights and even automotive headlights!

1.4 Resistive Circuits

1.4.1 Current Limiting Resistors

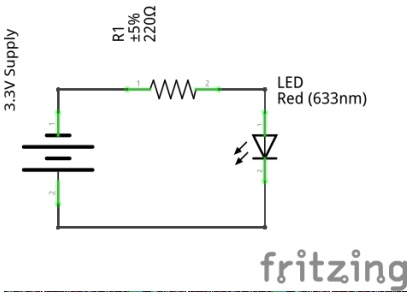
As a LED has very little resistance (when connected appropriately), when it is connected directly to a power supply, the current draw will exceed the specification of the LED and it will burn out.

$$V_{pp} - V_{LED} = IR \tag{1.3}$$

$$R >= \frac{V_{pp} - V_{LED}}{I_{max}} \tag{1.4}$$

For a 3.3V power supply, a 0.43V across the LED, and a max current of 100mA, the resistor needs to be greater than 29Ω.

Figure 1.7: Current Limiting Resistor



1.4.2 Series, Parallel, and More

1.4.3 Voltage Divider

So, what is a voltage divider

blah blah blah

and some more blah blah blah

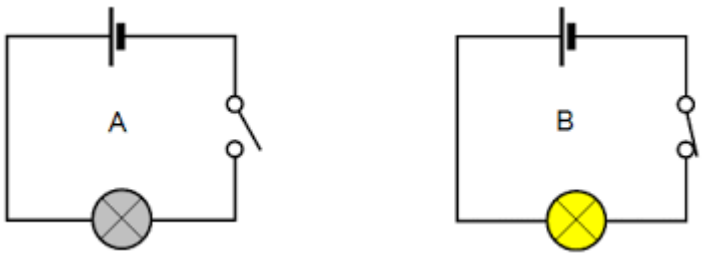
1.5 Switches

A switch is a component which controls the open-ness or closed-ness of an electric circuit. They allow control over current flow in a circuit (without having to actually get in there and manually cut or splice the wires). Switches are critical components in any circuit which requires user interaction or control.

A switch can only exist in one of two states: open or closed. In the off state (Figure 1.8 A), a switch looks like an open gap in the circuit. This, in effect, looks like an open circuit, preventing current from flowing.

In the on state (Figure 1.8 B), a switch acts just like a piece of perfectly-conducting wire. A short. This closes the circuit, turning the system "on" and allowing current to flow unimpeded through the rest of the system.

Figure 1.8: Switch States



1.5.1 Poles and Throws

A switch must have at least two terminals, one for the current to (potentially) go in, another to (potentially) come out. That only describes the simplest version of a switch though. More often than not, a switch has more than two pins. So how do all of those terminals line up with the internal workings of the switch? This is where knowing how many poles and throws a switch has is essential.

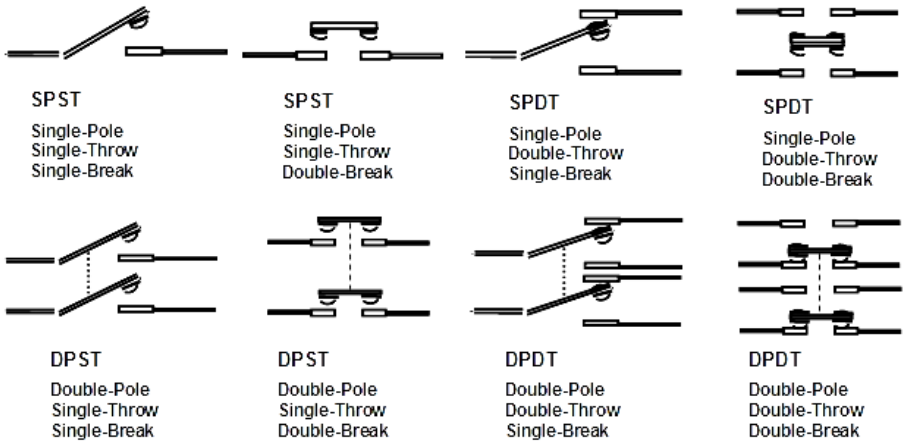
The number of poles* on a switch defines how many separate circuits the switch can control. So a switch with one pole, can only influence one single circuit. A four-pole switch can separately control four different circuits.

A switch's throw-count defines how many positions each of the switch's poles can be connected to. For example, if a switch has two throws, each circuit (pole) in the switch can be connected to one of two terminals.

Knowing how many poles and throws a switch has, it can be more specifically classified. Commonly you'll see switches defined as "single-pole, single-

throw", "single-pole, double-throw", "double-pole, double-throw", which are more often abbreviated down to SPST, SPDT, and DPDT, respectively.

Figure 1.9: Types of Switches



SPST

A single-pole, single-throw (SPST) switch is as simple as it gets. It's got one output and one input. The switch will either be closed or completely disconnected. SPSTs are perfect for on-off switching. They're also a very common form of momentary switches. SPST switches should only require two terminals.

SPDT

Another common switch-type is the SPDT. SPDTs have three terminals: one common pin and two pins which vie for connection to the common. SPDTs are great for selecting between two power sources, swapping inputs, or whatever it is you do with two circuits trying to go one place. Most simple slide switches are of the SPDT variety. SPDT switches should usually have

three terminals. (Sidenote: in a pinch an SPDT can actually be made into an SPST by just leaving one of the switch throws unconnected).

DPDT

Adding another pole to the SPDT creates a double-pole, double-throw (DPDT) switch. Basically two SPDT switches, which can control two separate circuits, but are always switched together by a single actuator. DPDTs should have six terminals.

1.5.2 Normally Open/Closed

When a momentary switch is not actuated, it's in a "normal" state. Depending on how the button is constructed, its normal state can be either an open circuit or a short circuit. When a button is open until actuated, it's said to be normally open (abbreviated NO). When you actuate an NO switch, you're closing the circuit, which is why these are also called "push-to-make" switches.

Conversely, if a button usually acts like a short circuit unless actuated, it's called a normally closed (NC) switch. NC switches are "push-to-break"; actuating the switch creates an open circuit.

Among the two types, you're probably much more likely to encounter a normally open momentary switch.

1.5.3 Momentary Switches

Momentary switches are switches which only remain in their on state as long as they're being actuated (pressed, held, magnetized, etc.). Most often momentary switches are best used for intermittent user-input cases; stuff like reset or keypad buttons.

Figure 1.10: Categories of Switches

Push-button

Push-button switches are the classic momentary switch. Typically these switches have a really nice, tactile, "clicky" feedback when you press them. They come in all sorts of flavors: big, small, colorful, illuminated (when an LED shines up through the button). They might be terminated as through-hole, surface-mount, or even panel-mount.

Others

Momentary switches don't always have to be actuated by a pushdown. It could be push-sideways, like the movement action in a handful of joysticks.

On the other end of the spectrum, reed switches open or close when exposed to the presence of a magnetic field. These are great for making a non-contact switch.

Figure 1.11: Reed Switch

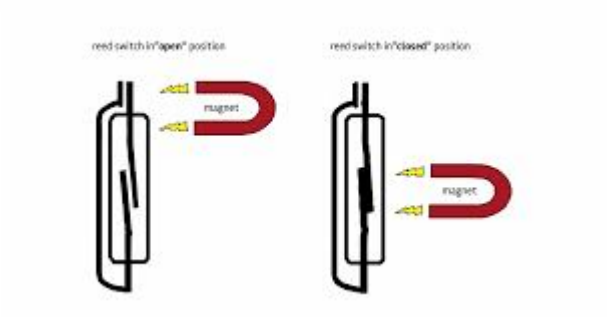


Figure 1.12: Example of a Reed Switch

1.5.4 Maintained Switches

A maintained switch retains its state until it's actuated into a new one. Just look to the nearest wall for an example of a maintained switch – the thing controlling your lights! Maintained switches are great for set-it-and-leave it applications like turning power on and off.

Slide Switch

Need a really basic, no-frills ON/OFF or selector switch. Slide switches might be for you! These switches have a tiny little nub which protrudes from the switch, and it slides across the body into one of two (or more) positions.

You'll usually find slide switches in SPDT or DPDT configurations. The common terminal is usually in the middle, and the two select positions are on the outside.

Toggle Switch

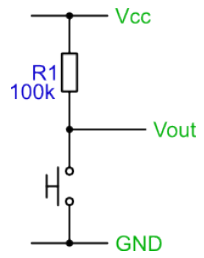
When you hear toggle switch, think "fire ze missiles!". Toggle switches have a long lever, which moves in a rocking motion. As they move to a new position, toggle switches make a really satisfying "snap".

1.5.5 Debounce

Figure 1.13 below shows a simple push switch with a pull-up resistor (the need for pull-up resistors will be described in Section 1.6.1). The right hand image shows the trace at the output terminal, V_{out}, when the switch is pressed. As can be seen, pressing the switch does not provide a clean edge. If this signal was used as an input to a digital counter, for example, you'd get multiple counts rather than the expected single count.

Note that the same can also occur on the release of a switch.

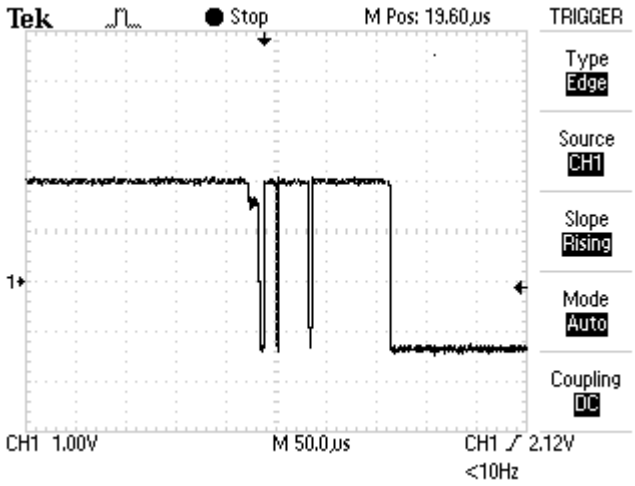
Figure 1.13: Pull Up Switch



The problem is that the contacts within the switch don't make contact cleanly, but actually slightly 'bounce'. The bounce is quite slow, so you can recreate the trace (see Figure 1.14), and the problem quite easily.

Code to remove the bounce (or debounce) can be found in Appendix A.1.1.

Figure 1.14: Switch Bounce Trace



1.5.6 Rotary Encoders

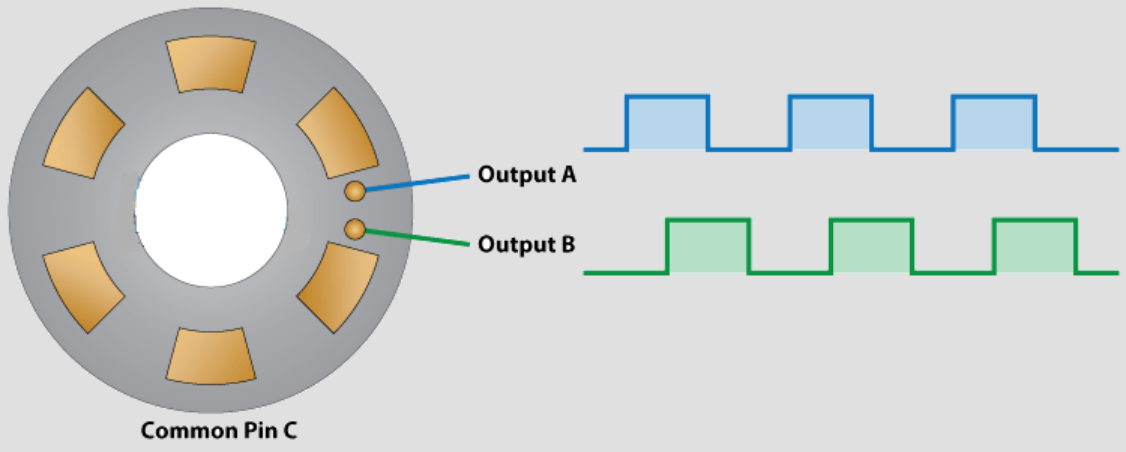
1.6 Digital and Analog Input/Output

1.6.1 Digital

Digital electronics rely on binary logic to store, process, and transmit data or information. Binary Logic refers to one of two states – ON or OFF. This is commonly translated as a binary 1 or binary 0. A binary 1 is also referred to as a HIGH signal and a binary 0 is referred to as a LOW signal.

Active-Low and Active-High When working with ICs and microcontrollers, you'll likely encounter pins that are active-low and pins that are active-high. Simply put, this just describes how the pin is activated. If it's an active-low pin, you must "pull" that pin LOW by connecting it to ground. For an active high pin, you connect it to your HIGH voltage (usually 3.3V/5V).

Figure 1.15: Rotary Encoder



TTL Logic Levels

A majority of systems we use rely on either 3.3V or 5 V TTL Levels. TTL is an acronym for Transistor-Transistor Logic. It relies on circuits built from bipolar transistors to achieve switching and maintain logic states. Transistors are basically fancy-speak for electrically controlled switches. For any logic family, there are a number of threshold voltage levels to know. Below is an example for standard 5V TTL levels, shown graphically in Figure **??**.

- V_{OH} – Minimum OUTPUT Voltage level a TTL device will provide for a HIGH signal.
- V_{IH} – Minimum INPUT Voltage level to be considered a HIGH.
- V_{OL} – Maximum OUTPUT Voltage level a device will provide for a LOW signal.
- V_{IL} – Maximum INPUT Voltage level to still be considered a LOW.

You will notice that the minimum output HIGH voltage (VOH) is 2.7 V. Basically, this means that output voltage of the device driving HIGH will

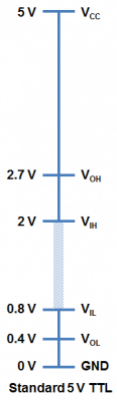


Figure 1.16: Voltage Levels Digital I/O

always be at least 2.7 V. The minimum input HIGH voltage (VIH) is 2 V, or basically any voltage that is at least 2 V will be read in as a logic 1 (HIGH) to a TTL device.

You will also notice that there is cushion of 0.7 V between the output of one device and the input of another. This is sometimes referred to as noise margin.

Likewise, the maximum output LOW voltage (VOL) is 0.4 V. This means that a device trying to send out a logic 0 will always be below 0.4 V. The maximum input LOW voltage (VIL) is 0.8 V. So, any input signal that is below 0.8 V will still be considered a logic 0 (LOW) when read into the device.

What happens if you have a voltage that is in between 0.8 V and 2 V? Well, your guess is as good as mine. Honestly, this range of voltages is undefined and results in an invalid state, often referred to as floating. If an output pin on your device is "floating" in this range, there is no certainty with what the signal will result in. It may bounce arbitrarily between HIGH and LOW.

Figure **??** is another way of looking at the input / output tolerances for a generic TTL device.

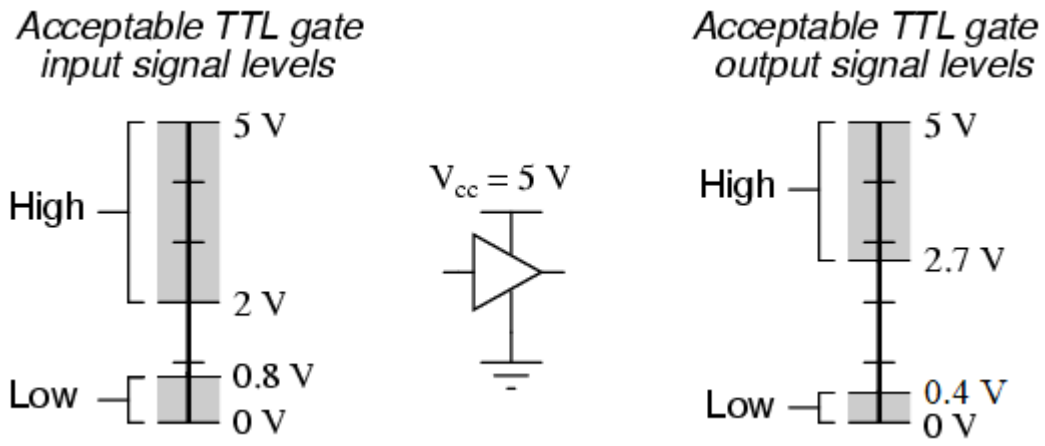


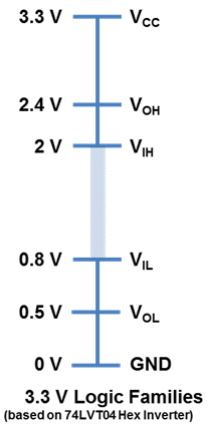
Figure 1.17: Voltage Levels TTL Digital I/O

CMOS Logic Levels

As technology has advanced, we have created devices that require lower power consumption and run off a lower base voltage (Vcc = 3.3 V instead of 5 V). The fabrication technique is also a bit different for 3.3 V devices that allows a smaller footprint and lower overall system costs.

In order to ensure general compatibility, you will notice that most of the voltage levels are almost all the same as 5 V devices. A 3.3 V device can interface with a 5V device without any additional components. For example, a logic 1 (HIGH) from a 3.3 V device will be at least 2.4 V. This will still be interpreted as a logic 1 (HIGH) to a 5V system because it is above the VIH of 2 V.

Figure 1.18: Voltage Levels CMOS Digital I/O



Warning A word of caution, however, is when going the other direction and interfacing from a 5 V to a 3.3 V device to ensure that the 3.3 V device is 5 V tolerant. The specification you are interested in is the maximum input voltage. On certain 3.3 V devices, any voltages above 3.6 V will cause permanent damage to the chip. You can use a simple voltage divider (like a 1KΩ and a 2KΩ) to knock down 5 V signals to 3.3 V levels or use one of our logic level shifters.

Pull Up/Down Reistor

Let's say you have an MCU with one pin configured as an input. If there is nothing connected to the pin and your program reads the state of the pin, will it be high (pulled to VCC) or low (pulled to ground)? It is difficult to tell. This phenomena is referred to as floating. To prevent this unknown state, a pull-up or pull-down resistor will ensure that the pin is in either a high or low state, while also using a low amount of current.

For simplicity, we will focus on pull-ups since they are more common than pull-downs. They operate using the same concepts, except the pull-up resistor is connected to the high voltage (this is usually 3.3V or 5V and is often

refereed to as VCC) and the pull-down resistor is connected to ground.

Pull-ups are often used with buttons and switches.

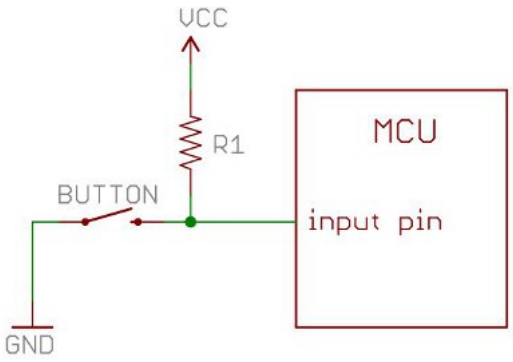


Figure 1.19: Pull Up Resistor

With a pull-up resistor, the input pin will read a high state when the button is not pressed. In other words, a small amount of current is flowing between VCC and the input pin (not to ground), thus the input pin reads close to VCC. When the button is pressed, it connects the input pin directly to ground. The current flows through the resistor to ground, thus the input pin reads a low state. Keep in mind, if the resistor wasn't there, your button would connect VCC to ground, which is very bad and is also known as a short.

Let's say you want to limit the current to approximately 1mA when the button is pressed in the circuit above, where Vcc = 5V. What resistor value should you use?

It is easy to show how to calculate the pull-up resistor using Ohm's Law:

$$V = I * R \tag{1.5}$$

Referring to the schematic above, Ohm's Law now is:

$$R_1 = \frac{V_{cc}}{CurrentThrouhgR_1} \tag{1.6}$$

Rearrange the above equation with some simple algebra to solve for the resistor:

$$R_1 = \frac{V_{cc}}{CurrentThrouhgR_1} = \frac{5V}{0.001A} = 5k\Omega \tag{1.7}$$

Remember to convert all of your units into volts, amps and Ohms before calculating (e.g. 1mA = 0.001 Amps). The solution is to use a 5k Ω resistor.

1.6.2 Analog

Input

An ADC, or Analog to Digital Converter, allows one to convert an analog voltage to a digital value that can be used by a microcontroller. There are many sources of analog signals that one might like to measure. There are analog sensors available that measure temperature, light intensity, distance, position, and force, just to name a few.

The ADC allows the ARM Cortex microcontroller to convert analog voltages to digital values with few to no external parts. The ARM Cortex features a 10-bit successive approximation ADC.

$$\frac{ResolutionofADC}{SystemVoltage} = \frac{ADCReading}{AnalogVoltageMeasured} \tag{1.8}$$

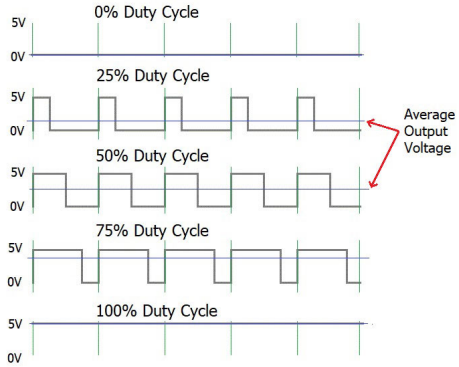
The way an ADC works is fairly complex. There are a few different ways to achieve this feat, but one of the most common technique uses the analog voltage to charge up an internal capacitor and then measure the time it takes to discharge across an internal resistor. The microcontroller monitors

the number of clock cycles that pass before the capacitor is discharged. This number of cycles is the number that is returned once the ADC is complete.

Output and Pulse Width Modulation (PWM)

Microcontrollers provide analog to digital (ADC) conversion to convert an input voltage to a digital value. You might think that they also provide the converse which is digital to analog (DAC) conversion. This is not the case. Instead they provide pulse-width modulated (PWM) outputs (see Figure 1.19). The Arduino library provides this functionality with a function called analogWrite(). The name seems to imply DAC functionality, but it just controls the PWM output. For many applications, such as the case of motor control, PWM is sufficient. For other applications, such as creating a linear voltage or current driver, a real DAC is needed.

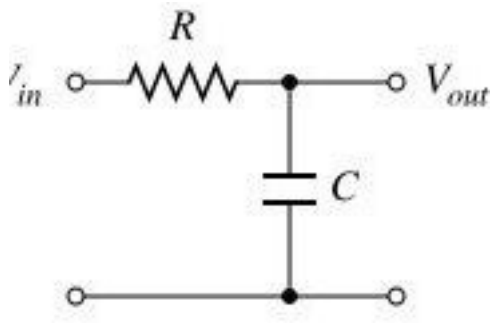
Figure 1.20: Setting Analog Output levels with PWM



Creating a real DAC

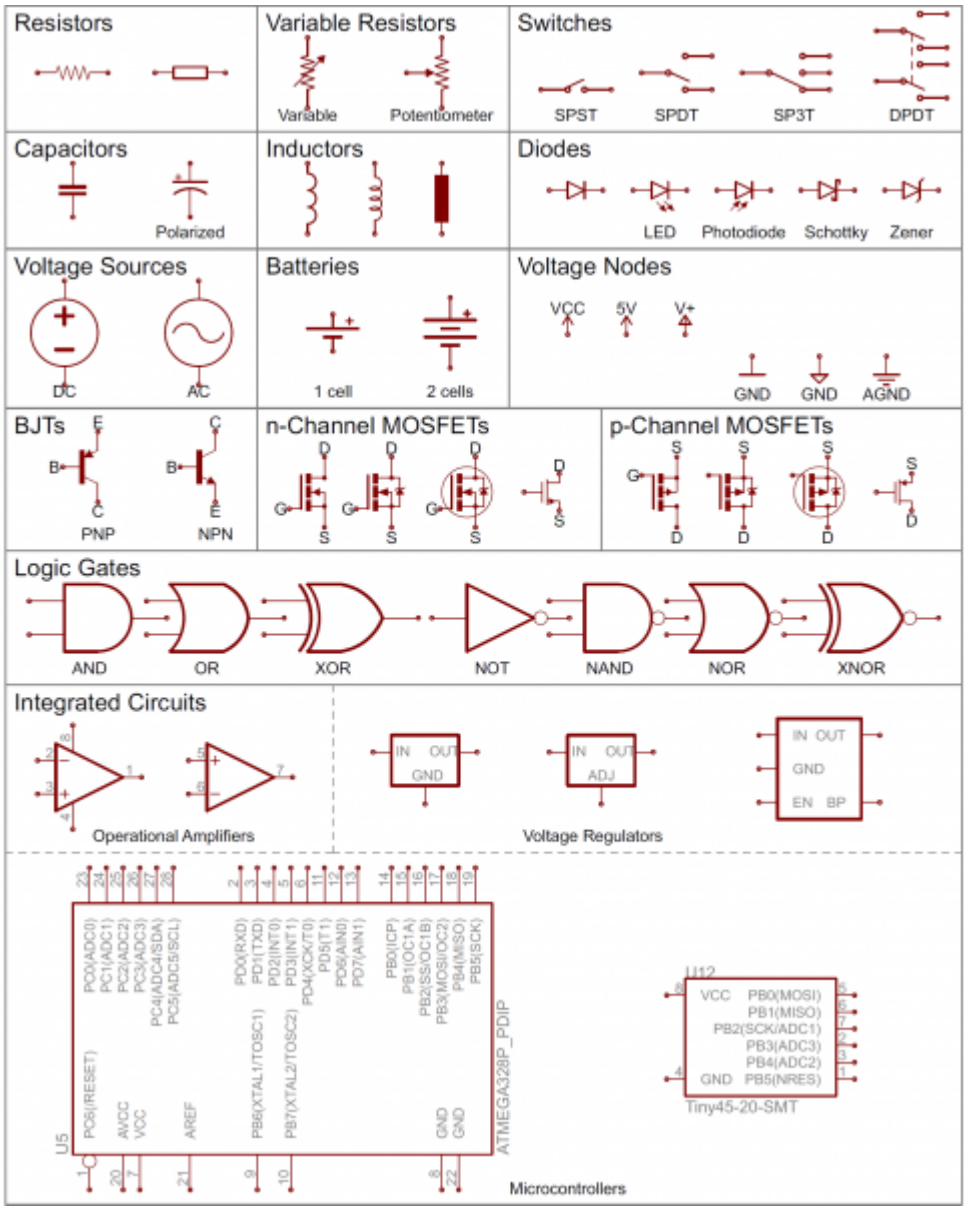
Fortunately, it is easy to convert a PWM output to an analog voltage level, producing a true DAC. All that is needed is a simple low-pass filter made from a resistor and a ceramic capacitor. The simple RC low-pass filter (see Figure 1.20) converts the PWM signal to a voltage proportional to the duty cycle. For the Arduino, an R value = 3.9K and a C value = 0.1uF works well for most applications.

Figure 1.21: Low Pass Filter to make true DAC



1.7 Reading Scematics

Figure 1.22: Typical Scematic Figures



```
from sys import argv

script, filename = argv

txt = open(filename)

print(f"Here's your file (filename):")
print(txt.read())

txt.close()

print("Type the filename again:")
file_again = input("> ")

txt_again = open(file_again)

print(txt_again.read())

txt_again.close()
```

Code Listing 1.1: Python code ex15.py

Chapter 2

Communication Protocols

2.1 Serial Communications

Embedded electronics is all about interlinking circuits (processors or other integrated circuits) to create a symbiotic system. In order for those individual circuits to swap their information, they must share a common communication protocol. Hundreds of communication protocols have been defined to achieve this data exchange, and, in general, each can be separated into one of two categories: parallel or serial.

Parallel vs. Serial Parallel interfaces transfer multiple bits at the same time. They usually require buses of data - transmitting across eight, sixteen, or more wires. Data is transferred in huge, crashing waves of 1's and 0's.

Parallel Generalization An 8-bit data bus, controlled by a clock, transmitting a byte every clock pulse. 9 wires are used. Serial interfaces stream their data, one single bit at a time. These interfaces can operate on as little as one wire, usually never more than four.

Serial Generalization Example of a serial interface, transmitting one bit every clock pulse. Just 2 wires required! Think of the two interfaces as a stream of cars: a parallel interface would be the 8+ lane mega-highway, while a serial interface is more like a two-lane rural country road. Over a set amount of

time, the mega-highway potentially gets more people to their destinations, but that rural two-laner serves its purpose and costs a fraction of the funds to build.

Parallel communication certainly has its benefits. It's fast, straightforward, and relatively easy to implement. But it requires many more input/output (I/O) lines. If you've ever had to move a project from a basic Arduino Uno to a Mega, you know that the I/O lines on a microprocessor can be precious and few. So, we often opt for serial communication, sacrificing potential speed for pin real estate.

Asynchronous Serial Over the years, dozens of serial protocols have been crafted to meet particular needs of embedded systems. USB (universal serial bus), and Ethernet, are a couple of the more well-known computing serial interfaces. Other very common serial interfaces include SPI, I2C, and the serial standard we're here to talk about today. Each of these serial interfaces can be sorted into one of two groups: synchronous or asynchronous.

A synchronous serial interface always pairs its data line(s) with a clock signal, so all devices on a synchronous serial bus share a common clock. This makes for a more straightforward, often faster serial transfer, but it also requires at least one extra wire between communicating devices. Examples of synchronous interfaces include SPI, and I2C.

Asynchronous means that data is transferred without support from an external clock signal. This transmission method is perfect for minimizing the required wires and I/O pins, but it does mean we need to put some extra effort into reliably transferring and receiving data. The serial protocol we'll be discussing in this tutorial is the most common form of asynchronous transfers. It is so common, in fact, that when most folks say "serial" they're talking about this protocol (something you'll probably notice throughout this tutorial).

The clock-less serial protocol we'll be discussing in this tutorial is widely used in embedded electronics. If you're looking to add a GPS module, Bluetooth, XBee's, serial LCDs, or many other external devices to your project, you'll probably need to whip out some serial-fu.

Now then, let's go on a serial journey...

Rules of Serial The asynchronous serial protocol has a number of built-in rules - mechanisms that help ensure robust and error-free data transfers. These mechanisms, which we get for eschewing the external clock signal, are:

Data bits, Synchronization bits, Parity bits, and Baud rate. Through the variety of these signaling mechanisms, you'll find that there's no one way to send data serially. The protocol is highly configurable. The critical part is making sure that both devices on a serial bus are configured to use the exact same protocols.

Baud Rate The baud rate specifies how fast data is sent over a serial line. It's usually expressed in units of bits-per-second (bps). If you invert the baud rate, you can find out just how long it takes to transmit a single bit. This value determines how long the transmitter holds a serial line high/low or at what period the receiving device samples its line.

Baud rates can be just about any value within reason. The only requirement is that both devices operate at the same rate. One of the more common baud rates, especially for simple stuff where speed isn't critical, is 9600 bps. Other "standard" baud are 1200, 2400, 4800, 19200, 38400, 57600, and 115200.

The higher a baud rate goes, the faster data is sent/received, but there are limits to how fast data can be transferred. You usually won't see speeds exceeding 115200 - that's fast for most microcontrollers. Get too high, and you'll begin to see errors on the receiving end, as clocks and sampling periods just can't keep up.

Framing the data Each block (usually a byte) of data transmitted is actually sent in a packet or frame of bits. Frames are created by appending synchronization and parity bits to our data.

Serial Packet A serial frame. Some symbols in the frame have configurable bit sizes. Let's get into the details of each of these frame pieces.

Data chunk The real meat of every serial packet is the data it carries. We ambiguously call this block of data a chunk, because its size isn't specifically stated. The amount of data in each packet can be set to anything from 5 to 9 bits. Certainly, the standard data size is your basic 8-bit byte, but other sizes have their uses. A 7-bit data chunk can be more efficient than 8, especially

if you're just transferring 7-bit ASCII characters.

After agreeing on a character-length, both serial devices also have to agree on the endianness of their data. Is data sent most-significant bit (msb) to least, or vice-versa? If it's not otherwise stated, you can usually assume that data is transferred least-significant bit (lsb) first.

Synchronization bits The synchronization bits are two or three special bits transferred with each chunk of data. They are the start bit and the stop bit(s). True to their name, these bits mark the beginning and end of a packet. There's always only one start bit, but the number of stop bits is configurable to either one or two (though it's commonly left at one).

The start bit is always indicated by an idle data line going from 1 to 0, while the stop bit(s) will transition back to the idle state by holding the line at 1.

Parity bits Parity is a form of very simple, low-level error checking. It comes in two flavors: odd or even. To produce the parity bit, all 5-9 bits of the data byte are added up, and the evenness of the sum decides whether the bit is set or not. For example, assuming parity is set to even and was being added to a data byte like 0b01011101, which has an odd number of 1's (5), the parity bit would be set to 1. Conversely, if the parity mode was set to odd, the parity bit would be 0.

Parity is optional, and not very widely used. It can be helpful for transmitting across noisy mediums, but it'll also slow down your data transfer a bit and requires both sender and receiver to implement error-handling (usually, received data that fails must be re-sent).

9600 8N1 (an example) 9600 8N1 - 9600 baud, 8 data bits, no parity, and 1 stop bit - is one of the more commonly used serial protocols. So, what would a packet or two of 9600 8N1 data look like? Let's have an example!

A device transmitting the ASCII characters 'O' and 'K' would have to create two packets of data. The ASCII value of O (that's uppercase) is 79, which breaks down into an 8-bit binary value of 01001111, while K's binary value is 01001011. All that's left is appending sync bits.

It isn't specifically stated, but it's assumed that data is transferred least-

significant bit first. Notice how each of the two bytes is sent as it reads from right-to-left.

"OK" Packet Since we're transferring at 9600 bps, the time spent holding each of those bits high or low is 1/(9600 bps) or 104 μs per bit.

For every byte of data transmitted, there are actually 10 bits being sent: a start bit, 8 data bits, and a stop bit. So, at 9600 bps, we're actually sending 9600 bits per second or 960 (9600/10) bytes per second.

Now that you know how to construct serial packets, we can move on to the hardware section. There we'll see how those 1's and 0's and the baud rate are implemented at a signal level!

Wiring and Hardware A serial bus consists of just two wires - one for sending data and another for receiving. As such, serial devices should have two serial pins: the receiver, RX, and the transmitter, TX.

Serial Wiring It's important to note that those RX and TX labels are with respect to the device itself. So the RX from one device should go to the TX of the other, and vice-versa. It's weird if you're used to hooking up VCC to VCC, GND to GND, MOSI to MOSI, etc., but it makes sense if you think about it. The transmitter should be talking to the receiver, not to another transmitter.

A serial interface where both devices may send and receive data is either full-duplex or half-duplex. Full-duplex means both devices can send and receive simultaneously. Half-duplex communication means serial devices must take turns sending and receiving.

Some serial busses might get away with just a single connection between a sending and receiving device. For example, our Serial Enabled LCDs are all ears and don't really have any data to relay back to the controlling device. This is what's known as simplex serial communication. All you need is a single wire from the master device's TX to the listener's RX line.

Hardware Implementation We've covered asynchronous serial from a conceptual side. We know which wires we need. But how is serial communication actually implemented at a signal level? In a variety of ways, actually. There

are all sorts of standards for serial signaling. Let's look at a couple of the more popular hardware implementations of serial: logic-level (TTL) and RS-232.

When microcontrollers and other low-level ICs communicate serially they usually do so at a TTL (transistor-transistor logic) level. TTL serial signals exist between a microcontroller's voltage supply range - usually 0V to 3.3V or 5V. A signal at the VCC level (3.3V, 5V, etc.) indicates either an idle line, a bit of value 1, or a stop bit. A 0V (GND) signal represents either a start bit or a data bit of value 0.

TTL Signal RS-232, which can be found on some of the more ancient computers and peripherals, is like TTL serial flipped on its head. RS-232 signals usually range between -13V and 13V, though the spec allows for anything from +/- 3V to +/- 25V. On these signals a low voltage (-5V, -13V, etc.) indicates either the idle line, a stop bit, or a data bit of value 1. A high RS-232 signal means either a start bit, or a 0-value data bit. That's kind of the opposite of TTL serial.

RS-232 Signal Between the two serial signal standards, TTL is much easier to implement into embedded circuits. However the low voltage levels are more susceptible to losses across long transmission lines. RS-232, or more complex standards like RS-485, are better suited to long range serial transmissions.

When you're connecting two serial devices together, it's important to make sure their signal voltages match up. You can't directly interface a TTL serial device with an RS-232 bus. You'll have to shift those signals!

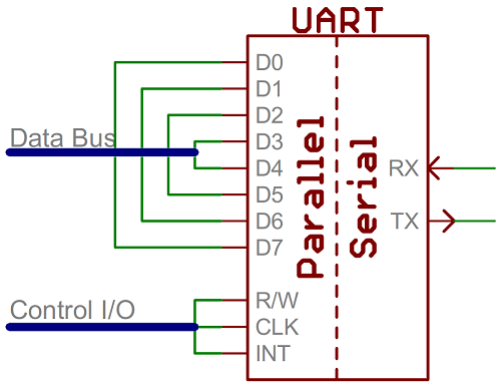
Continuing on, we'll explore the tool microcontrollers use to convert their data on a parallel bus to and from a serial interface. UARTs!

2.1.1 UART

The final piece to this serial puzzle is finding something to both create the serial packets and control those physical hardware lines. Enter the UART.

A universal asynchronous receiver/transmitter (UART) is a block of circuitry

Figure 2.1: Universal asynchronous receiver/transmitter (UART) circuitry



(see Figure **??**)responsible for implementing serial communication. Essentially, the UART acts as an intermediary between parallel and serial interfaces. On one end of the UART is a bus of eight-or-so data lines (plus some control pins), on the other is the two serial wires - RX and TX.

Common Mistakes

RX-to-TX, TX-to-RX
Seems simple enough, but it's a mistake I know I've made more than a few times. As much as you want their labels to match up, always make sure to cross the RX and TX lines between serial devices.

Baud Rate Mismatch
Baud rates are like the languages of serial communication. If two devices aren't speaking at the same speed, data can be either misinterpreted, or completely missed. If all the receiving device sees on its receive line is garbage, check to make sure the baud rates match up.

Bus Contention
Serial communication is designed to allow just two devices to communicate across one serial bus. If more than one device is trying to transmit on the same serial line you could run into bus-contention.

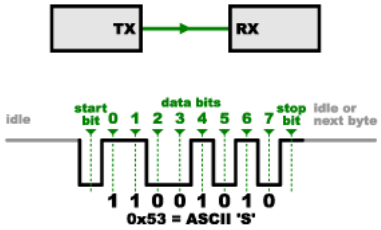
For example, if you're connecting a GPS module up to your Arduino, you may just wire that module's TX line up the Arduino's RX line. But that Arduino RX pin is already wired up to the TX pin of the USB-to-serial converter, which is used whenever you program the Arduino or use the Serial Monitor. This sets up the potential situation where both the GPS module and FTDI chip are trying to transmit on the same line at the same time.
Two devices trying to transmit data at the same time, on the same line, is bad! At "best" neither of the devices will get to send their data. At worst, both device's transmit lines go poof (though that's rare, and usually protected against).
It can be safe to connect multiple receiving devices to a single transmitting device. Not really up to spec and probably frowned upon by a hardened engineer, but it'll work. For example, if you're connecting a serial LCD up to an Arduino, the easiest approach may be to connect the LCD module's RX line to the Arduino's TX line. The Arduino's TX is already connected to the USB programmer's RX line, but that still leaves just one device in control of the transmission line.

2.1.2 Serial Peripherial Interface (SPI)

What's Wrong with Serial Ports? A common serial port, the kind with TX and RX lines, is called "asynchronous" (not synchronous) because there is no control over when data is sent or any guarantee that both sides are running at precisely the same rate. Since computers normally rely on everything being synchronized to a single "clock" (the main crystal attached to a computer that drives everything), this can be a problem when two systems with slightly different clocks try to communicate with each other.

To work around this problem, asynchronous serial connections add extra start and stop bits to each byte help the receiver sync up to data as it arrives. Both sides must also agree on the transmission speed (such as 9600 bits per second) in advance. Slight differences in the transmission rate aren't a problem because the receiver re-syncs at the start of each byte.

Figure 2.2: Asynchronous Serial Communications



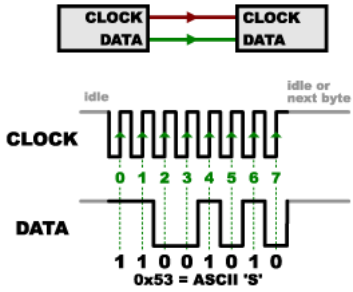
Asynchronous serial works just fine, but has a lot of overhead in both the extra start and stop bits sent with every byte, and the complex hardware required to send and receive data. And as you've probably noticed in your own projects, if both sides aren't set to the same speed, the received data will be garbage. This is because the receiver is sampling the bits at very specific times (the arrows in the above diagram). If the receiver is looking at the wrong times, it will see the wrong bits.

The Solution SPI works in a slightly different manner. It's a "synchronous" data bus, which means that it uses separate lines for data and a "clock" that keeps both sides in perfect sync. The clock is an oscillating signal that tells the receiver exactly when to sample the bits on the data line. This could be the rising (low to high) or falling (high to low) edge of the clock signal; the datasheet will specify which one to use. When the receiver detects that edge, it will immediately look at the data line to read the next bit (see the arrows in the below diagram). Because the clock is sent along with the data, specifying the speed isn't important, although devices will have a top speed at which they can operate (We'll discuss choosing the proper clock edge and speed in a bit).

Receiving Data You might be thinking to yourself, self, that sounds great for one-way communications, but how do you send data back in the opposite direction? Here's where things get slightly more complicated.

In SPI, only one side generates the clock signal (usually called CLK or SCK for Serial ClocK). The side that generates the clock is called the "master", and the other side is called the "slave". There is always only one master (which is almost always your microcontroller), but there can be multiple

Figure 2.3: Synchronous Serial Communications



slaves (more on this in a bit).

When data is sent from the master to a slave, it's sent on a data line called MOSI, for "Master Out / Slave In". If the slave needs to send a response back to the master, the master will continue to generate a prearranged number of clock cycles, and the slave will put the data onto a third data line called MISO, for "Master In / Slave Out".

Notice we said "prearranged" in the above description. Because the master always generates the clock signal, it must know in advance when a slave needs to return data and how much data will be returned. This is very different than asynchronous serial, where random amounts of data can be sent in either direction at any time. In practice this isn't a problem, as SPI is generally used to talk to sensors that have a very specific command structure. For example, if you send the command for "read data" to a device, you know that the device will always send you, for example, two bytes in return. (In cases where you might want to return a variable amount of data, you could always return one or two bytes specifying the length of the data and then have the master retrieve the full amount.)

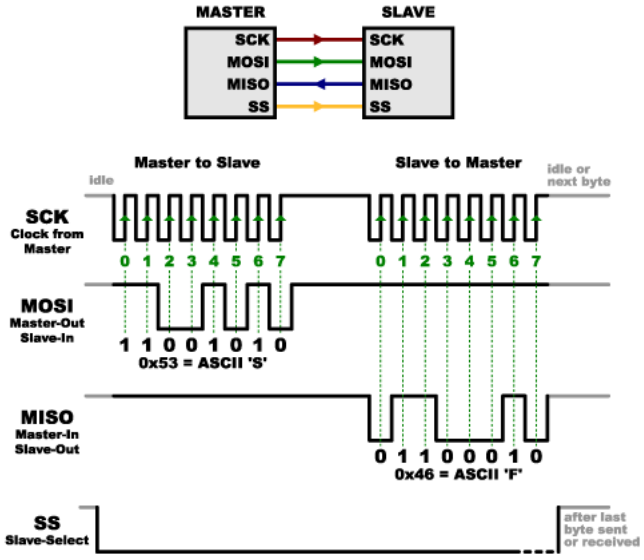
Note that SPI is "full duplex" (has separate send and receive lines), and, thus, in certain situations, you can transmit and receive data at the same time (for example, requesting a new sensor reading while retrieving the data from the previous one). Your device's datasheet will tell you if this is possible.

There's one last line you should be aware of, called SS for Slave Select. This tells the slave that it should wake up and receive / send data and is also used

when multiple slaves are present to select the one you'd like to talk to.

The SS line is normally held high, which disconnects the slave from the SPI bus. (This type of logic is known as "active low," and you'll often see used it for enable and reset lines.) Just before data is sent to the slave, the line is brought low, which activates the slave. When you're done using the slave, the line is made high again. In a shift register, this corresponds to the "latch" input, which transfers the received data to the output lines.

Figure 2.4: SPI Implementation



Multiple slaves There are two ways of connecting multiple slaves to an SPI bus:

In general, each slave will need a separate SS line. To talk to a particular slave, you'll make that slave's SS line low and keep the rest of them high (you don't want two slaves activated at the same time, or they may both try to talk on the same MISO line resulting in garbled data). Lots of slaves will require lots of SS lines; if you're running low on outputs, there are binary decoder chips that can multiply your SS outputs.

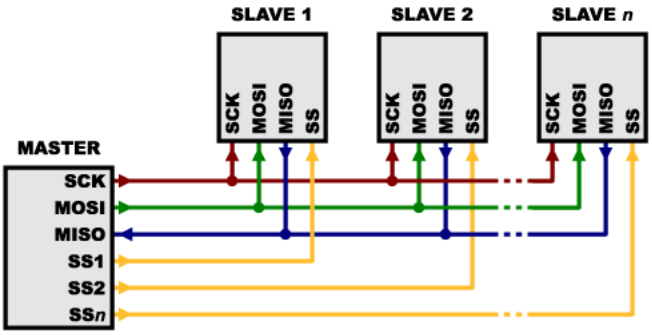
On the other hand, some parts prefer to be daisy-chained together, with the

MISO (output) of one going to the MOSI (input) of the next. In this case, a single SS line goes to all the slaves. Once all the data is sent, the SS line is raised, which causes all the chips to be activated simultaneously. This is often used for daisy-chained shift registers and addressable LED drivers.

Note that, for this layout, data overflows from one slave to the next, so to send data to any one slave, you'll need to transmit enough data to reach all of them. Also, keep in mind that the first piece of data you transmit will end up in the last slave.

This type of layout is typically used in output-only situations, such as driving LEDs where you don't need to receive any data back. In these cases you can leave the master's MISO line disconnected. However, if data does need to be returned to the master, you can do this by closing the daisy-chain loop (blue wire in the above diagram). Note that if you do this, the return data from slave 1 will need to pass through all the slaves before getting back to the master, so be sure to send enough receive commands to get the data you need.

Figure 2.5: SPI Multiple Devices

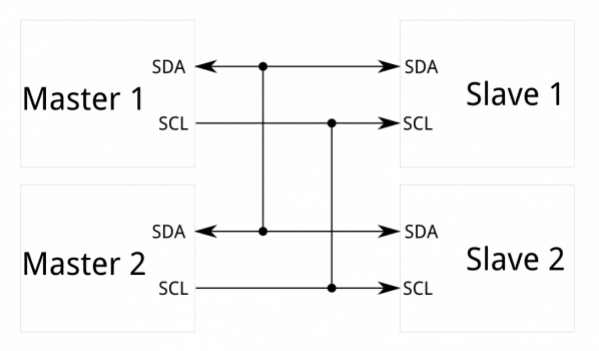


2.1.3 I2C

The Inter-integrated Circuit (I2C) Protocol is a protocol intended to allow multiple "slave" digital integrated circuits ("chips") to communicate with one or more "master" chips. Like the Serial Peripheral Interface (SPI), it is

only intended for short distance communications within a single device. Like Asynchronous Serial Interfaces (such as RS-232 or UARTs), it only requires two signal wires to exchange information.

Figure 2.6: I2C Communications



What's Wrong with Serial UART Ports?

Because serial ports are asynchronous (no clock data is transmitted), devices using them must agree ahead of time on a data rate. The two devices must also have clocks that are close to the same rate, and will remain so–excessive differences between clock rates on either end will cause garbled data.

Asynchronous serial ports require hardware overhead–the UART at either end is relatively complex and difficult to accurately implement in software if necessary. At least one start and stop bit is a part of each frame of data, meaning that 10 bits of transmission time are required for each 8 bits of data sent, which eats into the data rate.

Another core fault in asynchronous serial ports is that they are inherently suited to communications between two, and only two, devices. While it is possible to connect multiple devices to a single serial port, bus contention (where two devices attempt to drive the same line at the same time) is always an issue and must be dealt with carefully to prevent damage to the devices in question, usually through external hardware.

Finally, data rate is an issue. While there is no theoretical limit to asynchronous serial communications, most UART devices only support a certain

set of fixed baud rates, and the highest of these is usually around 230400 bits per second.

What's Wrong with SPI?

The most obvious drawback of SPI is the number of pins required. Connecting a single master to a single slave with an SPI bus requires four lines; each additional slave requires one additional chip select I/O pin on the master. The rapid proliferation of pin connections makes it undesirable in situations where lots of devices must be slaved to one master. Also, the large number of connections for each device can make routing signals more difficult in tight PCB layout situations. SPI only allows one master on the bus, but it does support an arbitrary number of slaves (subject only to the drive capability of the devices connected to the bus and the number of chip select pins available).

SPI is good for high data rate full-duplex (simultaneous sending and receiving of data) connections, supporting clock rates upwards of 10MHz (and thus, 10 million bits per second) for some devices, and the speed scales nicely. The hardware at either end is usually a very simple shift register, allowing easy implementation in software.

Enter I2C - The Best of Both Worlds! Block diagram of an I2C system I2C requires a mere two wires, like asynchronous serial, but those two wires can support up to 1008 slave devices. Also, unlike SPI, I2C can support a multi-master system, allowing more than one master to communicate with all devices on the bus (although the master devices can't talk to each other over the bus and must take turns using the bus lines).

Data rates fall between asynchronous serial and SPI; most I2C devices can communicate at 100kHz or 400kHz. There is some overhead with I2C; for every 8 bits of data to be sent, one extra bit of meta data (the "ACK/NACK" bit, which we'll discuss later) must be transmitted.

The hardware required to implement I2C is more complex than SPI, but less than asynchronous serial. It can be fairly trivially implemented in software.

I2C - A Brief History I2C was originally developed in 1982 by Philips for various Philips chips. The original spec allowed for only 100kHz communications,

and provided only for 7-bit addresses, limiting the number of devices on the bus to 112 (there are several reserved addresses, which will never be used for valid I2C addresses). In 1992, the first public specification was published, adding a 400kHz fast-mode as well as an expanded 10-bit address space. Much of the time (for instance, in the ATMega328 device on many Arduino-compatible boards) , device support for I2C ends at this point. There are three additional modes specified: fast-mode plus, at 1MHz; high-speed mode, at 3.4MHz; and ultra-fast mode, at 5MHz.

In addition to "vanilla" I2C, Intel introduced a variant in 1995 call "System Management Bus" (SMBus). SMBus is a more tightly controlled format, intended to maximize predictability of communications between support ICs on PC motherboards. The most significant difference between SMBus is that it limits speeds from 10kHz to 100kHz, while I2C can support devices from 0kHz to 5MHz. SMBus includes a clock timeout mode which makes low-speed operations illegal, although many SMBus devices will support it anyway to maximize interoperability with embedded I2C systems.

2.2 Internet Protocol

TCP/IP is a suite of protocols used by devices to communicate over the Internet and most local networks. It is named after two of it's original protocols—the Transmission Control Protocol (TCP) and the Internet Protocol (IP). TCP provides apps a way to deliver (and receive) an ordered and error-checked stream of information packets over the network. The User Datagram Protocol (UDP) is used by apps to deliver a faster stream of information by doing away with error-checking. When configuring some network hardware or software, you may need to know the difference.

Both TCP and UDP are protocols used for sending bits of data—known as packets—over the Internet. Both protocols build on top of the IP protocol. In other words, whether you're sending a packet via TCP or UDP, that packet is sent to an IP address. These packets are treated similarly, as they're forwarded from your computer to intermediary routers and on to the destination.

TCP and UDP aren't the only protocols that work on top of IP. However, they are the most widely used.

2.2.1 TCP

TCP is the most commonly used protocol on the Internet.

When you request a web page in your browser, your computer sends TCP packets to the web server's address, asking it to send the web page back to you. The web server responds by sending a stream of TCP packets, which your web browser stitches together to form the web page. When you click a link, sign in, post a comment, or do anything else, your web browser sends TCP packets to the server and the server sends TCP packets back.

TCP is all about reliability—packets sent with TCP are tracked so no data is lost or corrupted in transit. This is why file downloads don't become corrupted even if there are network hiccups. Of course, if the recipient is completely offline, your computer will give up and you'll see an error message saying it can't communicate with the remote host.

TCP achieves this in two ways. First, it orders packets by numbering them. Second, it error-checks by having the recipient send a response back to the sender saying that it has received the message. If the sender doesn't get a correct response, it can resend the packets to ensure the recipient receives them correctly.

2.2.2 UDP

The UDP protocol works similarly to TCP, but it throws out all the error-checking stuff. All the back-and-forth communication introduce latency, slowing things down.

When an app uses UDP, packets are just sent to the recipient. The sender doesn't wait to make sure the recipient received the packet—it just continues sending the next packets. If the recipient misses a few UDP packets here and

there, they are just lost—the sender won't resend them. Losing all this overhead means the devices can communicate more quickly.

UDP is used when speed is desirable and error correction isn't necessary. For example, UDP is frequently used for live broadcasts and online games.

For example, let's say you're watching a live video stream, which are often broadcast using UDP instead of TCP. The server just sends a constant stream of UDP packets to computers watching. If you lose your connection for a few seconds, the video may freeze or get jumpy for a moment and then skip to the current bit of the broadcast. If you experience minor packet-loss, the video or audio may be distorted for a moment as the video continues to play without the missing data.

This works similarly in online games. If you miss some UDP packets, player characters may appear to teleport across the map as you receive the newer UDP packets. There's no point in requesting the old packets if you missed them, as the game is continuing without you. All that matters is what's happening right now on the game server—not what happened a few seconds ago. Ditching TCP's error correction helps speed up the game connection and reduce latency.

Chapter 3

Coding

We are going to start off using the Arduino IDE[1]. The Arduino IDE is programmed essentailly using C++ code, but make the compiling and loading onto the microcontroller simplier.

We begin by installing the Arduino IDE: *https* : *//www.arduino.cc/en/main/software*

Then, we install the Teensyduino add-on: *https* : *//www.pjrc.com/teensy/td_download.html*

3.1 Number Systems used by computers

3.1.1 Bases

Number systems are the methods we use to represent numbers. Since grade school, we've all been mostly operating within the comfy confines of a base-10 number system, but there are many others. Base-2, base-8, base-16, base-20, base...you get the point. There are an infinite variety of base-number systems out there, but only a few are especially important to electrical engineering.

[1] An IDE, or Integrated Development Environment, enables programmers to consolidate the different aspects of writing a computer program.

The really popular number systems even have their own name. Base-10, for example, is commonly referred to as the decimal number system. Base-2, which we're here to talk about today, also goes by the moniker of binary. Another popular numeral system, base-16, is called hexadecimal.

The base of a number is often represented by a subscripted integer trailing a value. So in the introduction above, the first image would actually be 10010 somethings while the second image would be 100_2 somethings. This is a handy way to specify a number's base when there's ever any possibility of ambiguity.

3.1.2 Binary

Why binary you ask? Well, why decimal? We've been using decimal forever and have mostly taken for granted the reason we settled on the base-10 number system for our everyday number needs. Maybe it's because we have 10 fingers, or maybe it's just because the Romans forced it upon their ancient subjugates. Regardless of what lead to it, tricks we've learned along the way have solidified base-10's place in our heart; everyone can count by 10's. We even round large numbers to the nearest multiple of 10. We're obsessed with 10!

Computers and electronics are rather limited in the finger-and-toe department. At the lowest level, they really only have two ways to represent the state of anything: ON or OFF, high or low, 1 or 0. And so, almost all electronics rely on a base-2 number system to store, manipulate, and math numbers.

The heavy reliance electronics place on binary numbers means it's important to know how the base-2 number system works. You'll commonly encounter binary, or its cousins, like hexadecimal, all over computer programs. Analysis of Digital logic circuits and other very low-level electronics also requires heavy use of binary.

In this tutorial, you'll find that anything you can do to a decimal number can also be done to a binary number. Some operations may be even easier to do on a binary number (though others can be more painful).

3.1.3 Binary: Counting and Converting

The base of each number system is also called the radix. The radix of a decimal number is ten, and the radix of binary is two. The radix determines how many different symbols are required in order to flesh out a number system. In our decimal number system we've got 10 numeral representations for values between nothing and ten somethings: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. Each of those symbols represents a very specific, standardized value.

In binary we're only allowed two symbols: 0 and 1. But using those two symbols we can create any number that a decimal system can.

Counting in binary You can count in decimal endlessly, even in your sleep, but how would you count in binary? Zero and one in base-two should look pretty familiar: 0 and 1. From there things get decidedly binary.

Remember that we've only got those two digits, so as we do in decimal, when we run out of symbols we've got to shift one column to the left, add a 1, and turn all of the digits to right to 0. So after 1 we get 10, then 11, then 100. Let's start counting...

3.1.4 Converting from DEC to BIN

There's a handy function we can use to convert any binary number to decimal:

Binary to decimal conversion equation There are four important elements to that equation:

$a_n 2^n + a_{n-1} 2^{n-1} + ... + a_1 2^1 + a_0 2^0$

There are four important elements to that equation:

- a_n, a_{n-1}, a_1, etc., are the digits of a number. These are the 0's and 1's you're familiar with, but in binary they can only be 0 or 1.
- The position of a digit is also important to observe. The position starts at 0, on the right-most digit; this 1 or 0 is the least-significant. Every

Dec	Bin		Dec	Bin
0	0		16	10000
1	1		17	10001
2	10		18	10010
3	11		19	10011
4	100		20	10100
5	101		21	10101
6	110		22	10110
7	111		23	10111
8	1000		24	11000
9	1001		25	11001
10	1010		26	11010
11	1011		27	11011
12	1100		28	11100
13	1101		29	11101
14	1110		30	11110
15	1111		31	11111

Table 3.1: Decimal to Binary Conversion

digit you move to the left increases in significance, and also increases the position by 1.

- The length of a binary number is given by the value of n, actually it's n+1. For example, a binary number like 101 has a length of 3, something larger, like 10011110 has a length of 8.
- Each digit is multiplied by a weight: the 2^n, 2^{n-1}, 2^1, etc. The rightmost weight - 2^0 equates to 1, move one digit to the left and the weight becomes 2, then 4, 8, 16, 32, 64, 128, 256,... and on and on. Powers of two are of great importance to binary, they quickly become very familiar.

3.1.5 Bits, Nibbles, and Bytes

In discussing the make of a binary number, we briefly covered the length of the number. The length of a binary number is the amount of 1's and 0's it has.

Common binary number lengths Binary values are often grouped into a common length of 1's and 0's, this number of digits is called the length of a number. Common bit-lengths of binary numbers include bits, nibbles, and bytes (hungry yet?). Each 1 or 0 in a binary number is called a bit. From there, a group of 4 bits is called a nibble, and 8-bits makes a byte.

Bytes are a pretty common buzzword when working in binary. Processors are all built to work with a set length of bits, which is usually this length is a multiple of a byte: 8, 16, 32, 64, etc.

Length	Name	Example
1	Bit	0
4	Nibble	1011
8	Byte	10110101

Table 3.2: Bits, Nibbles, and Bytes

Word is another length buzzword that gets thrown out from time-to-time. Word is much less yummy sounding and much more ambiguous. The length of a word is usually dependent on the architecture of a processor. It could be 16-bits, 32, 64, or even more.

3.1.6 Hexidecimal Basics

Hexadecimal – also known as hex or base 16 – is a system we can use to write and share numerical values. In that way it's no different than the most famous of numeral systems (the one we use every day): decimal. Decimal is a base 10 number system (perfect for beings with 10 fingers), and it uses a collection of 10 unique digits, which can be combined to positionally represent numbers.

Hex, like decimal, combines a set of digits to create large numbers. It just so happens that hex uses a set of 16 unique digits. Hex uses the standard 0-9, but it also incorporates six digits you wouldn't usually expect to see creating numbers: A, B, C, D, E, and F.

Hex, along with decimal and binary, is one of the most commonly encountered numeral systems in the world of electronics and programming. It's important to understand how hex works, because, in many cases, it makes more sense to represent a number in base 16 than with binary or decimal.

3.1.7 Counting in HEX

Counting in hex is a lot like counting in decimal, except there are six more digits to deal with. Once a digit place becomes greater than "F", you roll that place over to "0", and increment the digit to the left by 1.

DEC	HEX		DEC	HEX		DEC	HEX		DEC	HEX
0	0		8	8		16	10		24	8
1	1		9	9		17	11		25	19
2	2		10	A		18	12		26	1A
3	3		11	B		19	13		27	1B
4	4		12	C		20	14		28	1C
5	5		13	D		21	15		29	1D
6	6		14	E		22	16		30	1E
7	7		15	F		23	17		31	1F

Table 3.3: Convert Decimal to Hexidecimal

3.1.8 HEX identifiers

"BEEF, it's what's for dinner". Am I channelling my inner Sam Elliott (McConaughey?), or expressing my hunger for the decimal number 48879? To avoid confusing situations like that, you'll usually see a hexadecimal number prefixed (or suffixed) with one of these identifiers:

Identifier	Example	Notes
0x	0x47DE	This prefix shows up a lot in UNIX C-based programming languages
#	#FF7734	Color references in HTML and image editting programs
%	%20	Often used in URLs to express characters like "Space" (%20)
\x	\x0A	Often used to express character control codes like "Backspace" (\x08), "Escape" (\x1B), and "Line Feed" (\x0A)
&#x	&#x3A9	Used in HTML, XML, and XHTML to express unicode characters (e.g. Ω prints an Ω).
0h	0h5E	A prefix used by many programmable graphic calculators (e.g. TI-89).
Numeral / Text Subscript	BE3716, 13Fhex	This is more of a mathematical representation of base 16 numbers. Decimal numbers can be represented with a subscript 10 (base 10). Binary is base 2.

Table 3.4: Hex Identifiers

There are a variety of other prefixes and suffixes that are specific to certain programming languages. Assembly languagues, for example, might use an "H" or "h" suffix (e.g. 7Fh) or a ""$prefix$(6AD). Consult examples if you're not sure which prefix or suffix to use with your programming language.

The "0x" prefix is one you'll see a lot, especially if you're doing any Arduino programming. We'll use that from now on in this tutorial.

In summary: DECAF? A horrible abomination of coffee. 0xDECAF? A perfectly acceptable, 5-digit hexadecimal number.

3.1.9 Converting Hex to Decimal

There's an ugly equation that rules over hex-to-decimal conversion:

$h_n 16^n + h_{n-1} 16^{n-1} + \cdots + h_1 16^1 + h_0 16^0$

There are a few important elements to this equation. Each of the h factors (hn, hn-1) is a single digit of the hex value. If our hex value is 0xF00D, for example, h0 is D, h1 and h2 are 0, and h3 is F.

Powers of 16 are a critical part of hexadecimal. More-signficant digits (those towards the left side of the number) are multiplied by larger powers of 16. The least-significant digit, h0, is multiplied by 160 (1). If a hex value is four digits long, the most-significant digit is multiplied by 163, or 4096.

To convert a hexadecimal number to decimal, you need to plug in values for each of the h factors in the equation above. Then multiply each digit by its respective power of 16, and add each product up. Our step-by-step approach is:

1. Start with the right-most digit of your hex value. Multiply it by 160, that is: multiply by 1. In other words, leave it be, but keep that value off to the side. [2]

[2]Remember to convert alphabetic hex values (A, B, C, D, E, and F) to their decimal equivalent (10, 11, 12, 13, 14, and 15)

```
1
2 // heading is used to document and include packages. It is
    also used to configure the coding environment.
3
4 void setup() {
5   // put your setup code here, to run once:
6
7 }
8
9 void loop() {
10  // put your main code here, to run repeatedly:
11
12 }
```

Code Listing 3.1: Basic Program Structure

2. Move one digit to the left. Multiply that digit by 161 (i.e. multipy by 16). Remember that product, and keep it to the side.

3. Move another digit left. Multiply that digit by 162 (256) and store that product.

4. Continue multiplying each incremental digit of the hex value by increasing powers of 16 (4096, 65536, 1048576, ...), and remember each product.

5. Once you've multiplied each digit of the hex value by the proper power of 16, add them all up. That sum is the decimal equivalent of your hex value.

3.2 Basic Program Structure

The basic code (as seen Code Listing 2.1) in has three sections: a heading, a set-up, and a loop. For ease of understanding, the code is color coded.

3.2.1 Comments

Comments are very important in your programs. They are used to tell you what something does in English, and they are used to disable parts of your program if you need to remove them temporarily.

There are a few ways to comment your code as seen in Code Listing 2.2. All comments are written in green in this tutorial; however, colors will vary depending on the IDE you are using.

3.2.2 Heading

3.2.3 Setup

setup() runs once, when the device is first turned on. Put initialization like pinMode and begin functions here.

3.2.4 loop

loop() runs over and over again, as quickly as it can execute. The core of your code will likely live here.

3.3 variables

3.3.1 data types

The Arduino environment is really just C++ with library support and built-in assumptions about the target environment to simplify the coding process. C++ defines a number of different data types; here we'll talk only about those used in Arduino with an emphasis on traps awaiting the unwary Arduino programmer.

```
/*
 * Project Comments
 * Description: Here is an example of how comments used in
   your code
 * Author: Brian Rashap
 * Date: 11/28/2019
 */

/* Everything between the slash-asterisk and the slash-
   asterisk
is seen by the compiler as a comment.
Even if
they cross
multiple
lines */

// a double slash can be used for a one line comment


void setup() { // everything after the double slash on the
   line is ignored

  /* comments can be used to explain what is going on in the
   code */
  // setup() runs once, when the device is first turned on.

}

void loop() {

  // loop() runs over and over again, as quickly as it can
   execute.

}
```

Code Listing 3.2: Comments

Below is a list of the data types commonly seen in Arduino, with the memory size of each in parentheses after the type name. Note: signed variables allow both positive and negative numbers, while unsigned variables allow only positive values.

- boolean (8 bit) - simple logical true/false
- byte (8 bit) - unsigned number from 0-255
- char (8 bit) - signed number from -128 to 127. The compiler will attempt to interpret this data type as a character in some circumstances, which may yield unexpected results
- unsigned char (8 bit) - same as 'byte'; if this is what you're after, you should use 'byte' instead, for reasons of clarity
- word (16 bit) - unsigned number from 0-65535
- unsigned int (16 bit)- the same as 'word'. Use 'word' instead for clarity and brevity
- int (16 bit) - signed number from -32768 to 32767. This is most commonly what you see used for general purpose variables in Arduino example code provided with the IDE
- unsigned long (32 bit) - unsigned number from 0-4,294,967,295. The most common usage of this is to store the result of the millis() function, which returns the number of milliseconds the current code has been running
- long (32 bit) - signed number from -2,147,483,648 to 2,147,483,647
- float (32 bit) - signed number from -3.4028235E38 to 3.4028235E38. Floating point on the Arduino is not native; the compiler has to jump through hoops to make it work. If you can avoid it, you should. We'll touch on this later.

3.3.2 variables

3.3.3 operators

3.3.4 pointers

In computer science, a pointer is a programming language object that stores the memory address of another value located in computer memory. A pointer references a location in memory, and obtaining the value stored at that location is known as dereferencing the pointer. As an analogy, a page number in a book's index could be considered a pointer to the corresponding page; dereferencing such a pointer would be done by flipping to the page with the given page number and reading the text found on that page. The actual format and content of a pointer variable is dependent on the underlying computer architecture.

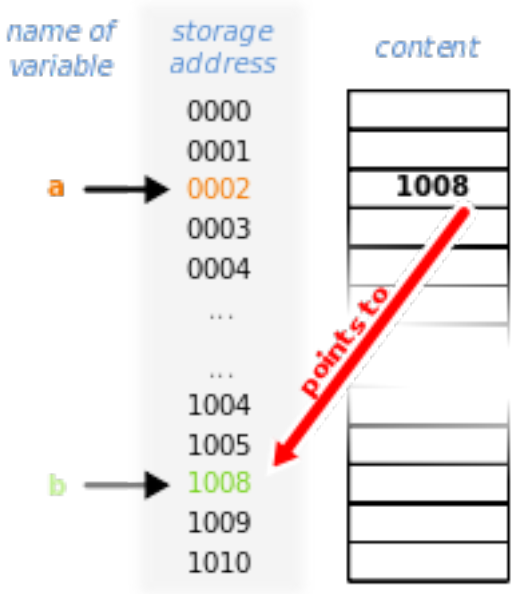


Figure 3.1: Pointers

Using pointers significantly improves performance for repetitive operations like traversing iterable data structures, e.g. strings, lookup tables, control tables and tree structures. In particular, it is often much cheaper in time and space to copy and dereference pointers than it is to copy and access the data to which the pointers point.

Pointers are also used to hold the addresses of entry points for called subroutines in procedural programming and for run-time linking to dynamic link libraries (DLLs). In object-oriented programming, pointers to functions are used for binding methods, often using what are called virtual method tables.

3.3.5 Pointer Syntax

The basic syntax to define a pointer is:

```
1 int *ptr;
```

This declares ptr as the identifier of an object of the following type:

- pointer that points to an object of type int

This is usually stated more succinctly as "ptr is a pointer to int."

Because the C language does not specify an implicit initialization for objects of automatic storage duration,[5] care should often be taken to ensure that the address to which ptr points is valid; this is why it is sometimes suggested that a pointer be explicitly initialized to the null pointer value, which is traditionally specified in C with the standardized macro NULL:[6]

```
1 int *ptr = NULL;
```

Dereferencing a null pointer in C produces undefined behavior,[7] which could be catastrophic. However, most implementations[citation needed] simply halt execution of the program in question, usually with a segmentation fault.

However, initializing pointers unnecessarily could hinder program analysis, thereby hiding bugs.

In any case, once a pointer has been declared, the next logical step is for it to point at something:

```
int a = 5;
int *ptr = NULL;
ptr = &a;
```

This assigns the value of the address of a to ptr. For example, if a is stored at memory location of 0x8130 then the value of ptr will be 0x8130 after the assignment. To dereference the pointer, an asterisk is used again:

```
*ptr = 8;
```

This means take the contents of ptr (which is 0x8130), "locate" that address in memory and set its value to 8. If a is later accessed again, its new value will be 8.

This example may be clearer if memory is examined directly. Assume that a is located at address 0x8130 in memory and ptr at 0x8134; also assume this is a 32-bit machine such that an int is 32-bits wide. The following is what would be in memory after the following code snippet is executed:

```
int a = 5;
int *ptr = NULL;
```

Address	Contents
0x8130	0x00000005
0x8134	0x00000000

(The NULL pointer shown here is 0x00000000.) By assigning the address of a to ptr:

```
ptr = &a;
```

yields the following memory values:

Address	Contents
0x8130	0x00000005
0x8134	0x00008130

Then by dereferencing ptr by coding:

```
1 *ptr = 8;
```

the computer will take the contents of ptr (which is 0x8130), 'locate' that address, and assign 8 to that location yielding the following memory:

Address	Contents
0x8130	0x00000008
0x8134	0x00008130

Clearly, accessing a will yield the value of 8 because the previous instruction modified the contents of a by way of the pointer ptr.

3.4 control statements

3.4.1 if-else and switch-case statements

3.4.2 for, while and do-while loops

3.5 others

3.5.1 pointers

3.5.2 arrays

3.5.3 structures

3.6 Serial Communications

3.6.1 Serial Tx/Rx

3.6.2 2 Wire (I2C)

See example in Section 6.3.

It find the addresses of I2C devices on the bus, you can use a program such as this:

```
/*
 * Project I2C_Scan
 * Description: Scan I2C bus and return found addresses
 * Author: Brian Rashap
 * Date: 13-Jan-2020
 */

#include "Particle.h"
```

```
9
10 unsigned long delayTime;
11
12 void setup()
13 {
14   Wire.begin();
15   Serial.begin(9600);
16   while(!Serial);    // time to get serial running
17   Serial.println("\nI2C Scanner");
18
19       unsigned status;
20
21     // default settings
22     // (you can also pass in a Wire library object like &
    Wire2)
23 }
24
25
26 void loop()
27 {
28   byte error, address;
29   int nDevices;
30
31   Serial.println("Scanning...");
32   delay(5000);
33
34
35   nDevices = 0;
36   for(address = 1; address < 127; address++ )
37   {
38     // The i2c_scanner uses the return value of
39     // the Write.endTransmisstion to see if
40     // a device did acknowledge to the address.
41     Wire.beginTransmission(address);
42     error = Wire.endTransmission();
43
44     if (error == 0)
45     {
46       Serial.print("I2C device found at address 0x");
47       if (address<16)
48         Serial.print("0");
49       Serial.print(address,HEX);
50       Serial.println("  !");
51
52       nDevices++;
```

```
53      }
54      else if (error==4)
55      {
56        Serial.print("Unknow error at address 0x");
57        if (address<16)
58          Serial.print("0");
59        Serial.println(address,HEX);
60      }
61    }
62    if (nDevices == 0)
63      Serial.println("No I2C devices found\n");
64    else
65      Serial.println("done\n");
66
67    delay(5000);           // wait 5 seconds for next scan
68  }
```

3.7 functions

Segmenting code into functions allows a programmer to create modular pieces of code that perform a defined task and then return to the area of code from which the function was "called". The typical case for creating a function is when one needs to perform the same action multiple times in a program.

Standardizing code fragments into functions has several advantages:

- Functions help the programmer stay organized. Often this helps to conceptualize the program.
- Functions codify one action in one place so that the function only has to be thought out and debugged once.
- This also reduces chances for errors in modification, if the code needs to be changed.
- Functions make the whole sketch smaller and more compact because sections of code are reused many times.

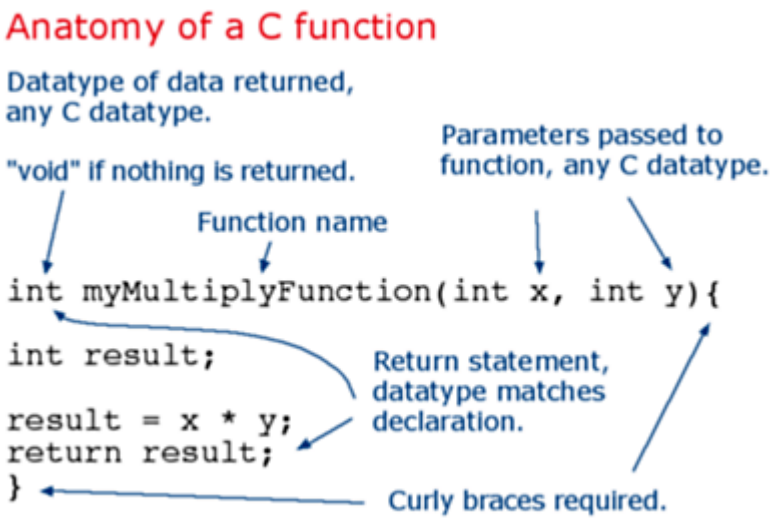


Figure 3.2: Anatomy of a C function

- They make it easier to reuse code in other programs by making it more modular, and as a nice side effect, using functions also often makes the code more readable.
- There are two required functions in an Arduino sketch, setup() and loop(). Other functions must be created outside the brackets of those two functions. As an example, we will create a simple function to multiply two numbers.

There are two required functions in an Arduino sketch, setup() and loop(). Other functions must be created outside the brackets of those two functions. As an example, we will create a simple function to multiply two numbers.

Our function needs to be declared outside any other function, so "myMultiplyFunction()" can go either above or below the "loop()" function.

To "call" our simple multiply function, we pass it parameters of the datatype that it is expecting. In our case below: k = myMultiplyFunction(i, j);

This function will read a sensor five times with analogRead() and calculate

```
void setup(){
  Serial.begin(9600);
}

void loop() {
  int i = 2;
  int j = 3;
  int k;

  k = myMultiplyFunction(i, j); // k now contains 6
  Serial.println(k);
  delay(500);
}

int myMultiplyFunction(int x, int y){
  int result;
  result = x * y;
  return result;
}
```

Code Listing 3.3: Ardunio Functions

the average of five readings. It then scales the data to 8 bits (0-255), and inverts it, returning the inverted result. As you can see, even if a function does not have parameters and no returns is expected "(" and ")" brackets plus ";" must be given.

```
void setup(){
  Serial.begin(9600);
}

void loop() {
  int sens;
  sens = ReadSens_and_Condition();
  Serial.println(sens);
  delay(500);
}



int ReadSens_and_Condition(){
  int i;
  int sval = 0;

  for (i = 0; i < 5; i++){
    sval = sval + analogRead(0);    // sensor on analog pin 0
  }

  sval = sval / 5;    // average
  sval = sval / 4;    // scale to 8 bits (0 - 255)
  sval = 255 - sval;  // invert output
  return sval;
}
```

Code Listing 3.4: Ardunio Functions

Chapter 4

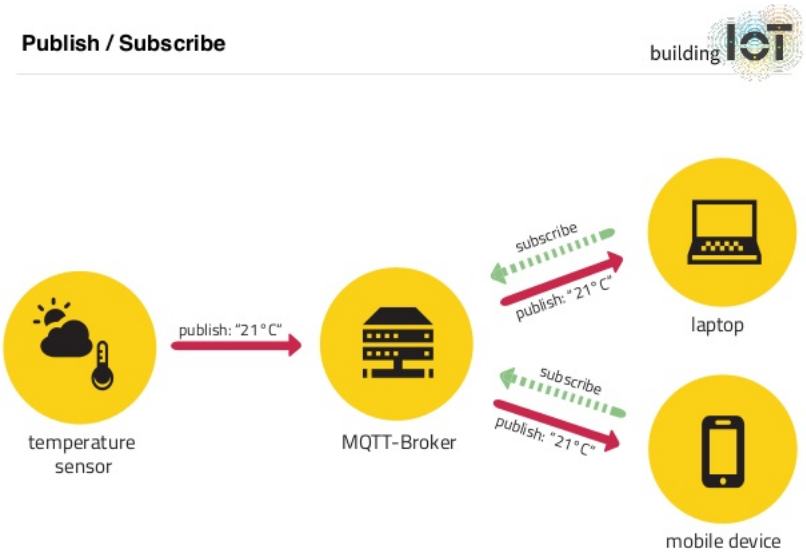
Wireless Communications

The Particle has an embedded ESP32 wireless coprocessor to connectivity to the Cloud.

4.1 MQTT

MQTT stands for MQ Telemetry Transport. It is a publish/subscribe, extremely simple and lightweight messaging protocol, designed for constrained devices and low-bandwidth, high-latency or unreliable networks. The design principles are to minimise network bandwidth and device resource requirements whilst also attempting to ensure reliability and some degree of assurance of delivery. These principles also turn out to make the protocol ideal of the emerging "machine-to-machine" (M2M) or "Internet of Things" world of connected devices, and for mobile applications where bandwidth and battery power are at a premium.

Figure 4.1: MQTT



4.2 Using MTQQ

4.2.1 Standard Ports

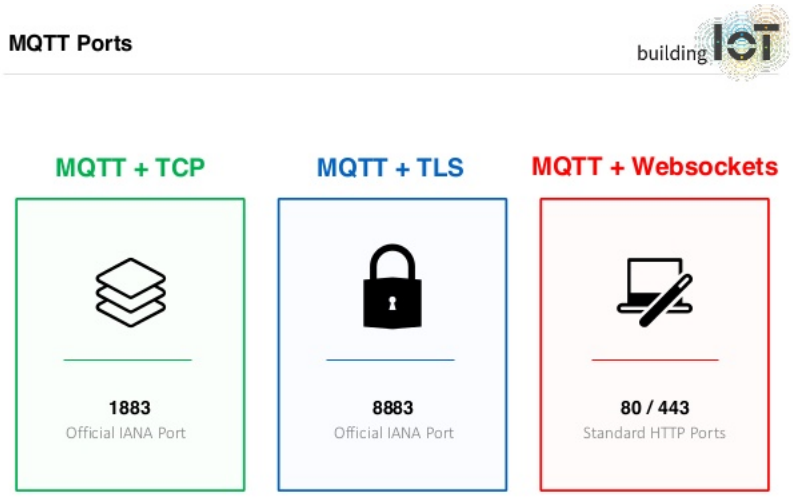
Are there standard ports for MQTT to use? Yes. TCP/IP port 1883 is reserved with IANA [1] for use with MQTT. TCP/IP port 8883 is also registered, for using MQTT over SSL.

4.2.2 MQTT security

You can pass a user name and password with an MQTT packet in V3.1 of the protocol. Encryption across the network can be handled with SSL,

[1] IANA is the Internet Assigned Numbers Authority which includes protocol assignments.

Figure 4.2: MQTT Sockets



independently of the MQTT protocol itself (it is worth noting that SSL is not the lightest of protocols, and does add significant network overhead). Additional security can be added by an application encrypting data that it sends and receives, but this is not something built-in to the protocol, in order to keep it simple and lightweight.

4.3 Adafruit IO

4.3.1 Using MQTT with Adafruit IO

You will want to use the following details to connect a MQTT client to Adafruit IO:

- Host: *io.adafruit.com*

- Port: 1883 (or 8883 for SSL encrypted connection)
- Username: your Adafruit account username (see the accounts.adafruit.com page here to find yours)
- Password: your Adafruit IO key (click the AIO Key button on a dashboard to find the key)

It is strongly recommend using SSL if your MQTT client allows it.

If the MQTT library requires that you set a client ID then use a unique value like a random GUID. Most MQTT libraries handle setting the client ID to a random value automatically though.

Adafruit IO's MQTT API exposes feed data using special topics. You can publish a new value for a feed to its topic, or you can subscribe to a feed's topic to be notified when the feed has a new value. Any one of the following topic forms is valid for a feed:

(username)/feeds/(feed name or key)
(username)/f/(feed name or key)

Where (username) is your Adafruit IO username (the same as specified when connecting to the MQTT server) and (feed name or key) is the feed's name or key. The smaller '/f/' path is provided as a convenience for small embedded clients that need to save memory.

In order to publish to Adafruit IO, you simply use

4.4 Particle Cloud

4.4.1 Using JSOM with ThingSpeak.com

```
#include <Adafruit_MQTT.h>

#include "Adafruit_MQTT/Adafruit_MQTT.h"
#include "Adafruit_MQTT/Adafruit_MQTT_SPARK.h"

/***** Adafruit.io Setup *****************/
#define AIO_SERVER      "io.adafruit.com"
#define AIO_SERVERPORT  1883  // use 1883 or 8883 for SSL
#define AIO_USERNAME  "<insert username>"
#define AIO_KEY       "<insert AI Key>"

/*** Global State (you don't need to change this!) ***/
TCPClient TheClient;

// Setup the MQTT client class by passing in the WiFi client
// and MQTT server and login details.
Adafruit_MQTT_SPARK mqtt(&TheClient,AIO_SERVER,AIO_SERVERPORT
    ,AIO_USERNAME,AIO_KEY);

/***** Feeds *******************************/
// Notice MQTT paths for AIO follow the form:
// <username>/feeds/<feedname>

Adafruit_MQTT_Publish <feedname1> = Adafruit_MQTT_Publish(&
    mqtt, AIO_USERNAME "/feeds/<feedname1>");
Adafruit_MQTT_Publish <feedname2> = Adafruit_MQTT_Publish(&
    mqtt, AIO_USERNAME "/feeds/<feedname2>");
Adafruit_MQTT_Publish <feedname3> = Adafruit_MQTT_Publish(&
    mqtt, AIO_USERNAME "/feeds/<feedname3>");
```

Code Listing 4.1: Setting up MQTT

```
if(mqtt.Update()) {
    gy_x.publish(x);
    gy_y.publish(y);
    gy_z.publish(z);
    }
```

Code Listing 4.2: Publishing to Adafruit IO using MQTT

Chapter 5

Hardware

5.1 Teensy 3.2

5.2 Creating header files for Arduino IDE

If the include file is part of a single Sketch, instructions are here... http://www.arduino.cc/en/Hackin

If the include file is meant to be shared by multiple Sketches then...

1. Close the Arduino IDE
2. Navigate to the Arduino/libraries directory
3. Create a subdirectory. I suggest something like MyCommon.
4. In the new subdirectory, create a header file. I suggest something like colors.h.
5. Open the new header file, edit it as you wish, and save it
6. Open the Arduino IDE
7. Create or open a Sketch

8. Add a #include to the top of the Sketch that references your new include file

5.3 Particle Argon

5.3.1 Overview

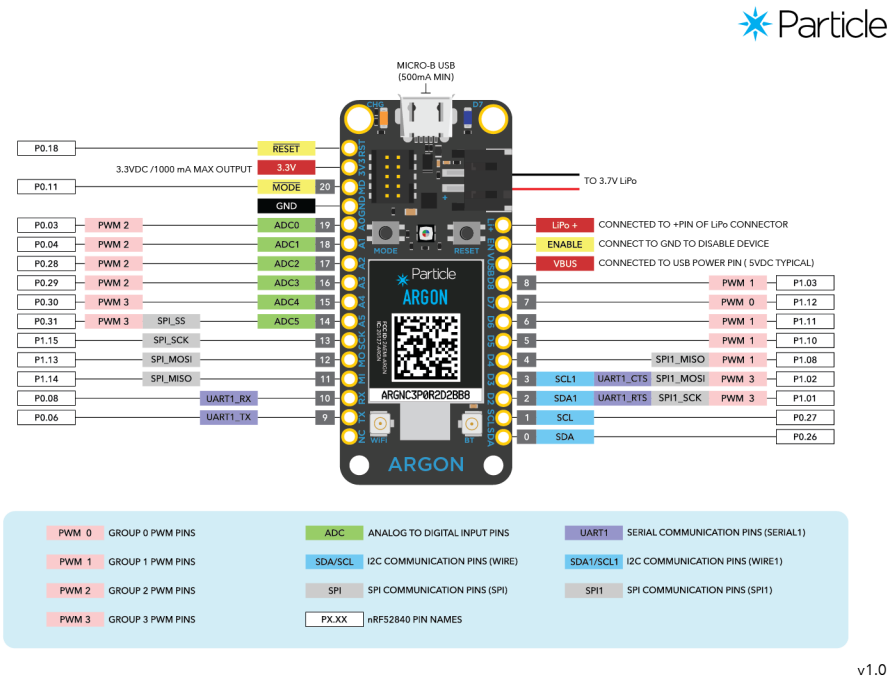
The Argon is a powerful Wi-Fi enabled development board that can act as either a standalone Wi-Fi endpoint or Wi-Fi enabled gateway for Particle Mesh networks. It is based on the Nordic nRF52840 and has built-in battery charging circuitry so it's easy to connect a Li-Po and deploy your local network in minutes.

The Argon is great for connecting existing projects to the Particle Device Cloud or as a gateway to connect an entire group of local endpoints.

5.3.2 Features

- Espressif ESP32-D0WD 2.4 GHz Wi-Fi coprocessor
  - On-board 4MB flash for ESP32
  - 802.11 b/g/n (2.4 GHz), up to 150 Mbps
- Nordic Semiconductor nRF52840 SoC
  - ARM Cortex-M4F 32-bit processor @ 64MHz
  - 1MB flash, 256KB RAM
  - IEEE 802.15.4-2006 (e.g. Zigbee): 250 Kbps
  - Bluetooth 5: 2 Mbps, 1 Mbps, 500 Kbps, 125 Kbps
  - NFC-A tag
- On-board additional 4MB SPI flash
- 20 mixed signal GPIO (6 x Analog, 8 x PWM), UART, I2C, SPI

Figure 5.1: Particle Argon Pin Layout



- Micro USB 2.0 full speed (12 Mbps)
- Integrated Li-Po charging and battery connector
- JTAG (SWD) Connector

Pin	Description
Li+	Positive terminal of the LiPo battery connector.
VUSB	USB (+ve) supply.
3V3	Output of the on-board 3.3V regulator.
GND	System ground pin.
EN	Device enable pin: To disable the device, connect this pin to GND.
RST	Active-low system reset input. This pin is internally pulled-up.
MD	Connected to the MODE button. The MODE function is active-low.
RX	UART RX, but can also be used as a digital GPIO.
TX	UART TX, but can also be used as a digital GPIO.
SDA	Data pin for I2C, but can also be used as a digital GPIO.
SCL	Clock pin for I2C, but can also be used as a digital GPIO.
MO,MI,SCK	SPI interface pins, but can also be used as a digital GPIO.
D2-D8	Generic GPIO pins. D2-D8 are PWM-able.
A0-A5	Analog input pins, also digital GPIO. A0-A5 are PWM-able.

Table 5.1: Particle Argon Pin Layout

Figure 5.2: Particle Argon Block Diagram

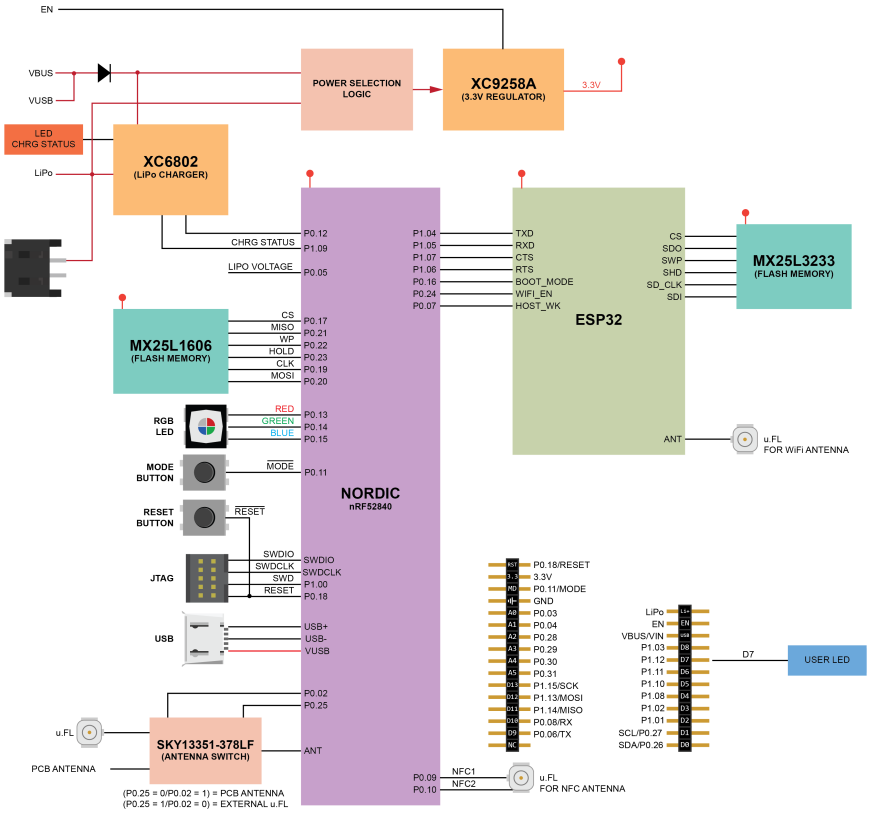


Figure 5.3: Particle Argon Pin Markings

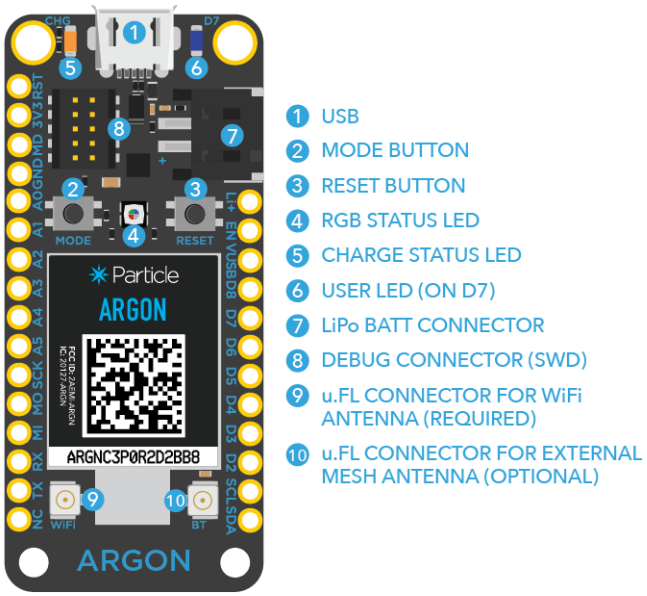
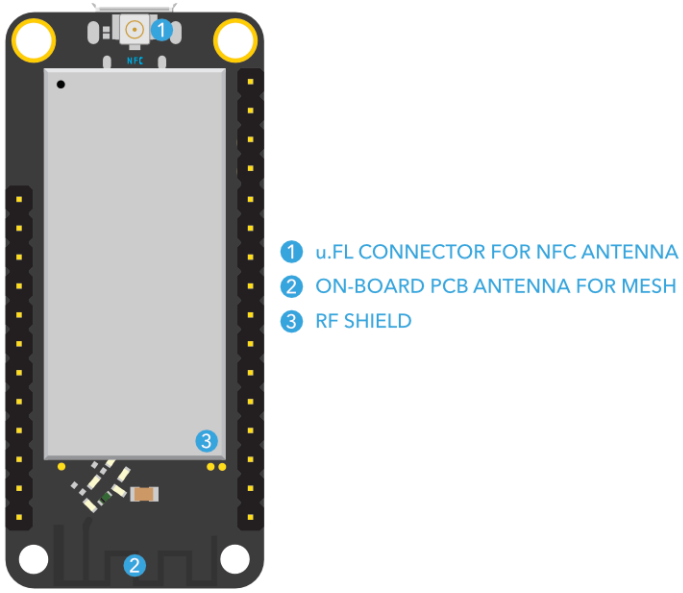


Figure 5.4: Particle Argon Bottom Pin Markings



Mode	LED Status
Connected	Breathing Cyan
OTA Firmware Update	Blinking Magenta (red and blue together)
Looking for Internet	Blinking Green
Connecting to Cloud	Rapid Blinking Cyan
Listening Mode	Blinking Blue
Network Reset	Rapid Blinking Blue
WiFi Off	Breathing White
Safe Mode	Breathing Magenta (red and blue together)
DFU (Device Firmware Upgrade	Blinking Yellow
Restore Factory Firmware	Rapid Blinking Yellow
Factory Reset	Rapid Blinking White
Decryption Error	Blinking Cyan followed by 1 Orange Blink
No Internet	Blinking Cyan followed by 2 Orange Blink
No Particle Cloud	Blinking Cyan followed by 3 Orange Blink
Authentication Error	Blinking Cyan followed by 1 Magenta Blink
Handshake Error	Blinking Cyan followed by 1 Red Blink
Decryption Error	Blinking Cyan followed by 1 Orange Blink
SOS - Firmware Crash	Blinking Red - 3 short, 3 long, 3 short, error code

Table 5.2: Particle Argon Modes - LED Sequence

Chapter 6

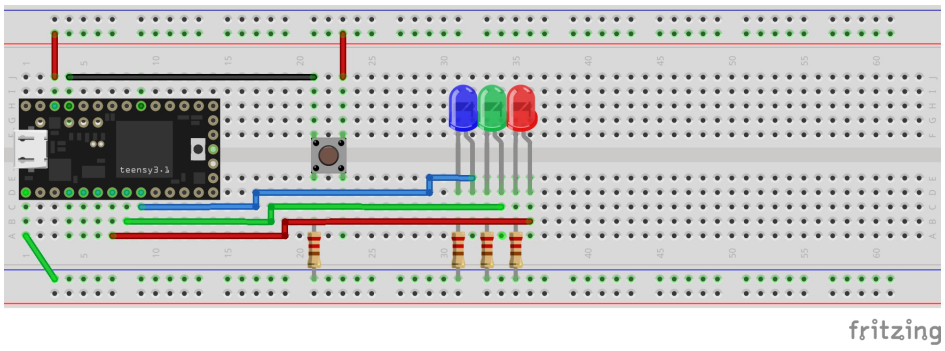
Projects

6.1 OneButton Library

In this project we will learn how to use the OneButton library to control multiple LEDs with one button. In addition, we'll introduce the concept of Arrays.

The bread board layout can be seen in Figure 6.1.

Figure 6.1: oneButtonLED Breadboard layout



The schematic is seen in Figure 6.2 and the PCB layout in Figure 6.3

Figure 6.2: oneButtonLED Scematic

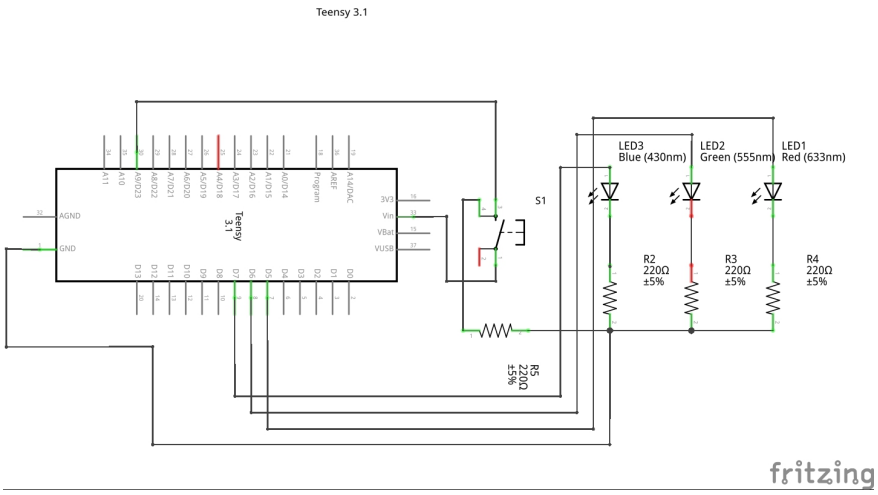
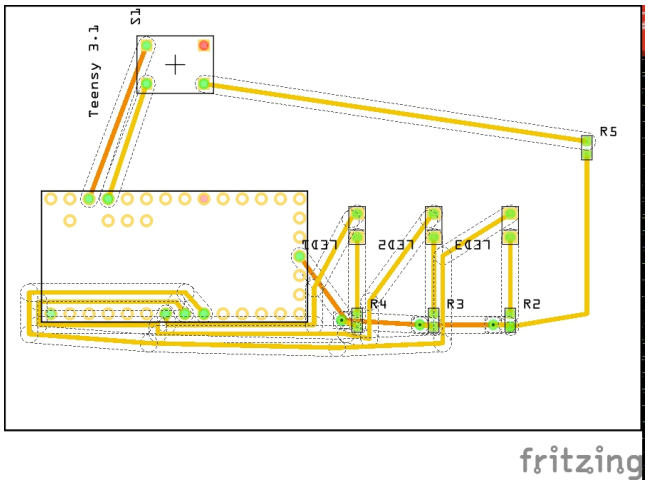


Figure 6.3: oneButtonLED Printed Circuit Board



The Code for running this project, including the Array that has the LED pins is here.

```
/*
 * Project: Using OneButton library - introducing an array
 * Description: Introduct Students to OneButton Library using
    1 button and an array
```

```
 * Author: Brian A Rashap
 * Date: 13-Dec-2019
 */

#include "OneButton.h"

// Setup OneButton on pin 23
OneButton button1(23, false);

// LED pin configure
int ledPin = 5;
int ledPin2 = 6;
int ledPin3 = 7;
int ledArray[] = {ledPin, ledPin2, ledPin3};

boolean buttonState = LOW;

int n = sizeof(ledArray);      // counter for the Array
int j = 0;                     // variable for which LED to
   light

void setup() {

  Serial.begin(9600);
  while (!Serial);

  pinMode(ledPin, OUTPUT);
  pinMode(ledPin2, OUTPUT);
  pinMode(ledPin3, OUTPUT);

  for (n = 0; n<3; n++) {
    Serial.print(ledArray[n]);
    Serial.print(" , ");
  }
  Serial.println();

  // link the button 1 functions.
  button1.attachClick(click1);
  button1.attachDoubleClick(doubleclick1);
  button1.attachLongPressStart(longPressStart1);
  button1.attachLongPressStop(longPressStop1);
  //button1.attachDuringLongPress(longPress1);
  button1.setClickTicks(250);
  button1.setPressTicks(2000);

```

```
48   if(Serial) Serial.println("Starting oneButtonArray...");
49
50 }
51
52 void loop() {
53   // keep watching the push buttons:
54   button1.tick();
55 }
56
57
58 // ----- button 1 callback functions
59
60 // This function will be called when the button1 was pressed
       1 time (and no 2. button press followed).
61 void click1() {
62   Serial.println("Button 1 click.");
63   buttonState = !buttonState;
64   Serial.print("buttonState = ");
65   Serial.println(buttonState);
66   if (buttonState == LOW) digitalWrite(ledArray[j], LOW);
67   else digitalWrite(ledArray[j], HIGH);
68
69 } // click1
70
71
72 // This function will be called when the button1 was pressed
       2 times in a short timeframe.
73 void doubleclick1() {
74   Serial.println("Button 1 doubleclick.");
75   j++;
76   if (j > 2) j = 0;
77   Serial.print("j = ");
78   Serial.println(j);
79 } // doubleclick1
80
81
82 // This function will be called once, when the button1 is
       pressed for a long time.
83 void longPressStart1() {
84   Serial.println("Button 1 longPress start");
85   for (n = 0; n<3; n++) {
86     digitalWrite(ledArray[n], HIGH);
87     delay(200);
88     digitalWrite(ledArray[n], LOW);
89   }
```

```
90 } // longPressStart1
91
92
93 // This function will be called often, while the button1 is
       pressed for a long time.
94 void longPress1() {
95   Serial.println("Button 1 longPress...");
96 } // longPress1
97
98
99 // This function will be called once, when the button1 is
       released after beeing pressed for a long time.
100 void longPressStop1() {
101   Serial.println("Button 1 longPress stop");
102   for (n = 2; n>=0; n--) {
103     digitalWrite(ledArray[n], HIGH);
104     delay(200);
105     digitalWrite(ledArray[n], LOW);
106   }
107 } // longPressStop1
```

6.2 GPS reciever using GPS6MV2

We are going to learn how to use Serial Communications by creating a simple GPS unit. For this project we will be using the GPS6MV2 GPS transceiver. For UART Serial communications we connect the sending serial pin to the respective receiivng pin. For the Teensy 3.2 we connect the TX1 (Pin1) to the RX Pin on the GPS6MV2 unit and RX1 (Pin0) to the TX Pin.

The bread board layout can be seen in Figure 6.4.

The scematic is seen in Figure 6.5 and the PCB layout in Figure 6.6

Figure 6.4: GPS Breadboard layout

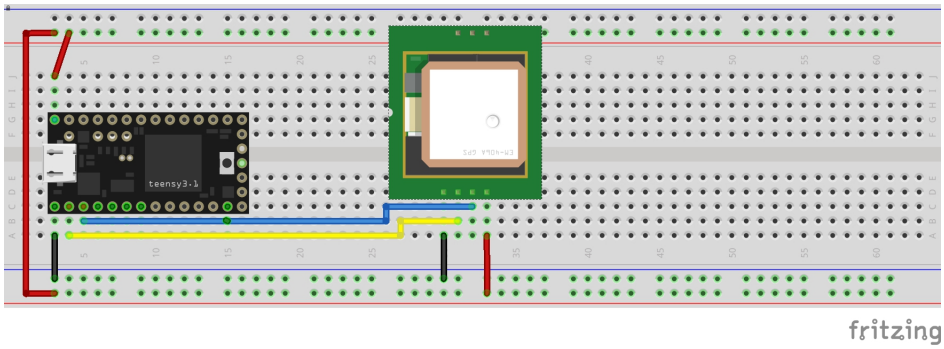
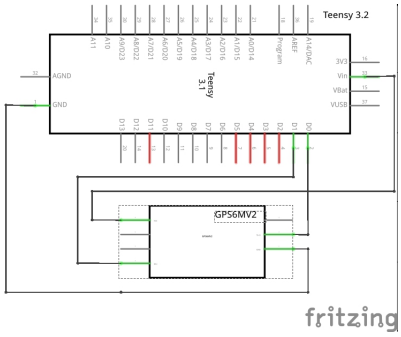


Figure 6.5: GPS Scematic



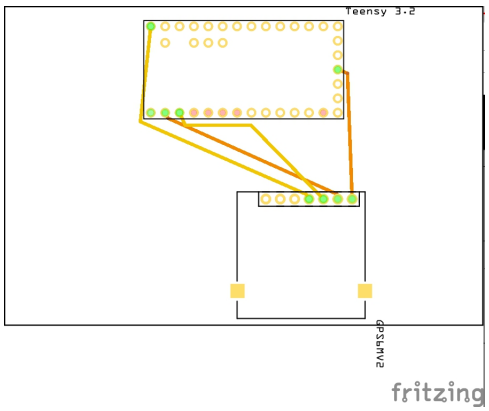
6.3 NCD ConnectEverything

```
/*
 * Project:      NCD0x50
 * Description:  Using the NCD I2C mini modules
 *               More info found at ControlEverything.com
 * Author:       Brian Rashap
 * Date:         9-Jan-2020
 */

#include <Wire.h>
#include <application.h>
#include <spark_wiring_i2c.h>

// ADC121C_MQ131 I2C address is 0x50(80)
```

Figure 6.6: GPS Printed Circuit Board




```
14 #define Addr1 0x50   // Ozone Sensor
15 #define Addr3 0x52   // CO2 Sensor
16
17 int raw_adc = 0;
18 double ppm = 0.0;
19
20 void setup()
21 {
22   // Set variable
23   Particle.variable("i2cdevice131", "ADC121C_MQ131");
24   Particle.variable("i2cdevice9", "ADC121C_MQ9");
25   Particle.variable("PPM", ppm);
26
27   // Initialise I2C communication as MASTER
28   Wire.begin();
29   // Initialise serial communication, set baud rate = 9600
30   Serial.begin(9600);
31   delay(300);
32 }
33
34 void loop()
35 {
36   oz();
37   co2();
38   delay(60000);
39 }
40
41 void oz() {
```

```
42      unsigned int data[2];
43
44    // Start I2C transmission
45    Wire.beginTransmission(Addr1);
46    // Select data register
47    Wire.write(0x00);
48    // Stop I2C transmission
49    Wire.endTransmission();
50
51    // Request 2 bytes of data
52    Wire.requestFrom(Addr1, 2);
53
54    // Read 2 bytes of data
55    // raw_adc msb, raw_adc lsb
56    if (Wire.available() == 2)
57    {
58      data[0] = Wire.read();
59      data[1] = Wire.read();
60    }
61
62    // Convert the data to 12-bits
63    raw_adc = ((data[0] & 0x0F) * 256) + data[1];
64    ppm = (1.99 * raw_adc) / 4095.0 + 0.01;
65
66    // Output data to dashboard
67    //Particle.publish("Ozone Concentration : ", String(ppm) +
       " ppm");
68    Particle.publish("Ozone", String(ppm));
69    Serial.print("Ozone Concenration (ppm): ");
70    Serial.println(ppm);
71  }
72
73  void co2()
74  {
75    unsigned int data[2];
76
77    // Start I2C transmission
78    Wire.beginTransmission(Addr3);
79    // Select data register
80    Wire.write(0x00);
81    // Stop I2C transmission
82    Wire.endTransmission();
83
84    // Request 2 bytes of data
85    Wire.requestFrom(Addr3, 2);
```

```
86
87    // Read 2 bytes of data
88    // raw_adc msb, raw_adc lsb
89    if (Wire.available() == 2)
90    {
91      data[0] = Wire.read();
92      data[1] = Wire.read();
93    }
94    delay(300);
95
96    // Convert the data to 12-bits
97    raw_adc = ((data[0] & 0x0F) * 256) + data[1];
98    ppm = (1000 / 4096.0) * raw_adc + 10;
99
100   // Output data to dashboard
101   // Particle.publish("Carbon Monoxide Concentration : ",
       String(ppm) + " ppm");
102   Particle.publish("CO2", String(ppm));
103   Serial.print("CO2 Concenration (ppm): ");
104   Serial.println(ppm);
105 }
```

Appendix A

Example Code

A.1 Useful Code Segments

A.1.1 Switch Debounce

```
const int buttonPin = A1; // the pushbutton pin
const int ledPin = 13;    // the LED pin

int ledState = HIGH;         // output state to LED
int buttonState;             // input current reading
int lastButtonState = HIGH;  // input previous reading

// 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.

// the last time the output pin was toggled
unsigned long lastDebounceTime = 0;
// the debounce time; increase if the output flickers
unsigned long debounceDelay = 50;

void setup() {
  pinMode(buttonPin, INPUT_PULLUP);
  pinMode(ledPin, OUTPUT);

```

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

void loop() {
  // read the state of the switch
  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) {
    // if 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 for next time through the loop:
  lastButtonState = reading;
}
```

A.1.2 Synchronizing Time Function (Particle)

```
void sync_my_time() {
  Time.zone(-7); // Set Time Zone to Mountain (UTC - 7)
  unsigned long cur = millis();

    // Request time synchronization from Particle Cloud
    Particle.syncTime();

    // Wait until Argon receives time from Particle Cloud
    // (or connection to Particle Device Cloud is lost)
    waitUntil(Particle.syncTimeDone);
    // Check if synchronized successfully
    if (Particle.timeSyncedLast() >= cur)
    {
      // Print current time
      Serial.print("The current time (MST) is: ");
      Serial.println(Time.timeStr());
    }
}
```

A.2 GPS code using TinyGPS++

Here is the Arduino code to run the GPS. The available GPS read commands can be found in Table A.1.

```
#include <TinyGPS++.h>

/*
 * Project: GPS Serial Communication
 * Description: Interface with GPS6MV2 using Teensy UART
   serial communication
 * Author: Brian A Rashap
 * Date: 15-Oct-19
 */

// Define gps as an TingGPSPlus object
TinyGPSPlus gps;

/* On Teensy, the UART (real serial port) is always best to
   use. */
/* Unlike Arduino, there's no need to use NewSoftSerial
   because */
```

```
/* the "Serial" object uses the USB port, leaving the UART
   free. */
/* Select Serial1 for Uart, alterntively, Serial2 or Serial3
   could be selected */
/* For Serial1 use Pin 0 for Rx and Pin 1 for Tx */

#define Uart Serial1
int hour;

void setup()
{
  Serial.begin(115200);
  Uart.begin(9600);
  delay(1000);
}

void loop()
{
  while (Uart.available() > 0)  {
    char c = Uart.read();
    Serial.print(c);  // Print raw GPS data
    gps.encode(c);
    }

  if (gps.altitude.isUpdated()) {
    Serial.println();
    Serial.println("Acquiring Data");
    Serial.println("--------- ----");
    Serial.print("LAT = "); Serial.println(gps.location.lat()
    , 6); // Latitude in degrees (double)
    Serial.print("LON = "); Serial.println(gps.location.lng()
    , 6); // Longitude in degrees (double)
    Serial.print("SPD = "); Serial.println(gps.speed.mps());
    // Speed in meters per second (double)
    Serial.print("ALT = "); Serial.println(gps.altitude.
    meters()); // Altitude in meters (double)
    Serial.print("SAT = "); Serial.println(gps.satellites.
    value()); // Number of satellites in use (u32)
    Serial.print("PER = "); Serial.println(gps.hdop.value());
    // Horizontal Dim. of Precision (100ths-i32)

    // GPS Time is Univeral Time
    hour = gps.time.hour();
    hour = hour - 6; //convert to Mountain Standard Time
    if (hour < 0) {
```

```
52          hour = hour + 24;
53      }
54      Serial.print("TIME = ");
55      Serial.print(hour); Serial.print(":"); // Hour (0-23) (u8
        )
56      Serial.print(gps.time.minute()); Serial.print(":"); //
        Minute (0-59) (u8)
57      Serial.println(gps.time.second()); // Second (0-59) (u8)
58      Serial.println();
59      delay(5000); // Wait 5 second for next reading
60    }
61 }
```

A.3 Sensor published to Adafruit IO

Here is the code to connect a BME280 Temperature/Pressure/Humidty sensor to the Paricle Argon and publish results to the Adafruit IO Dashboard. See Chapter B for details on Adafruit IO.

```
1  /*
2   * Project BME280Dashboard
3   * Description: BME280 Data to Adafruit IO cloud
4   * Note: As with all I2C projects, I include a
5   *       I2C Scan function to validate that the
6   *       I2C connections are working
7   * Author: Brian Rashap
8   * Date: 11/7/2019
9   */
10
11 #include <Adafruit_MQTT.h>
12
13
14 #include "Adafruit_MQTT/Adafruit_MQTT.h"
15 #include "Adafruit_MQTT/Adafruit_MQTT_SPARK.h"
16
17 /************************ Adafruit.io Setup
       ***********************************/
18 #define AIO_SERVER      "io.adafruit.com"
19 #define AIO_SERVERPORT  1883                   // use 8883
       for SSL
20 #define AIO_USERNAME    "<Insert username>"
21 #define AIO_KEY         "<Insert Adafruit IO Key>"
```

TingGPS++ Parameter	Description
gps.location.lat()	Latitude in degrees (double)
gps.location.lng()	Longitude in degrees (double)
gps.speed.mps()	Speed in meters per second (double)
gps.altitude.meters()	Altitude in meters (double)
gps.satellites.value()	Number of satellites in use (u32)
gps.hdop.value()	Hor. Dim. of Precision (100ths-i32)
gps.location.rawLat().negative	"-" : "+");
gps.location.rawLat().deg	Raw latitude in whole degrees
gps.location.rawLat().billionths	... and billionths (u16/u32)
gps.location.rawLng().negative	"-" : "+");
gps.location.rawLng().deg	Raw longitude in whole degrees
gps.location.rawLng().billionths)	... and billionths (u16/u32)
gps.time.value()	Raw time in HHMMSSCC format (u32)
gps.date.value()	Raw date in DDMMYY format (u32)
gps.date.year()	Year (2000+) (u16)
gps.date.month()	Month (1-12) (u8)
gps.date.day()	Day (1-31) (u8)
gps.time.hour()	Hour (0-23) (u8)
gps.time.minute()	Minute (0-59) (u8)
gps.time.second()	Second (0-59) (u8)
gps.time.centisecond()	100ths of a second (0-99) (u8)
gps.speed.value()	Raw speed in 100ths of a knot (i32)
gps.speed.knots()	Speed in knots (double)
gps.speed.mph()	Speed in miles per hour (double)
gps.speed.kmph()	Speed in kilometers per hour (double)
gps.course.value()	Raw course in 100ths of a degree (i32)
gps.course.deg()	Course in degrees (double)
gps.altitude.value()	Raw altitude in centimeters (i32)
gps.altitude.miles()	Altitude in miles (double)
gps.altitude.kilometers()	Altitude in kilometers (double)
gps.altitude.feet()	Altitude in feet (double)

Table A.1: TinyGPS++ available parameters

22

```
23 /************ Global State (you don't need to change this!)
       ***   ***************/
24 TCPClient TheClient;
25
26 // Setup the MQTT client class by passing in the WiFi client
       and MQTT server and login details.
27 Adafruit_MQTT_SPARK mqtt(&TheClient,AIO_SERVER,AIO_SERVERPORT
       ,AIO_USERNAME,AIO_KEY);
28
29 /****************************** Feeds
       ***************************************/
30 // Setup a feed called 'voltage' for publishing.
31 // Notice MQTT paths for AIO follow the form: <username>/
       feeds/<feedname>
32
33 Adafruit_MQTT_Publish temp = Adafruit_MQTT_Publish(&mqtt,
       AIO_USERNAME "/feeds/Temperature");
34 Adafruit_MQTT_Publish press = Adafruit_MQTT_Publish(&mqtt,
       AIO_USERNAME "/feeds/Pressure");
35 Adafruit_MQTT_Publish rh = Adafruit_MQTT_Publish(&mqtt,
       AIO_USERNAME "/feeds/Relative_Humidity");
36
37 /************************** Setup BME280
       ***********************************/
38
39 #include "Particle.h"
40 #include "Adafruit_Sensor.h"
41 #include "Adafruit_BME280.h"
42
43 Adafruit_BME280 bme;
44
45 unsigned long delayTime;
46 #define SEALEVELPRESSURE_HPA (1013.25)
47
48 /************************** Sketch Code
       ************************************/
49 float temperature;
50 float pressure;
51 float humidity;
52
53 void setup() {
54     Wire.begin();
55
56     Serial.begin(115200);
```

```
57      while(!Serial);         // wait for serial to start
        running
58      Serial.println("\nI2C Scanner");
59
60      unsigned status;
61
62      status = bme.begin();
63      if (!status) {
64          Serial.println("Could not find a valid BME280 sensor,
        check writing");
65          while (1);
66      }
67      delayTime = 1000;
68      I2CScan();
69  }
70
71  void loop()
72  {
73      printValues();
74      temperature = bme.readTemperature()*1.6+32.0;
75      pressure = bme.readPressure() / 100.0;
76      humidity = bme.readHumidity();
77
78      Serial.println("-----------------Data to Publish
        ------------------");
79      Serial.println(temperature);
80      Serial.println(pressure);
81      Serial.println(humidity);
82      Serial.println("------------------End Publish
        --------------------");
83
84
85  if(mqtt.Update()) {
86      temp.publish(temperature);
87      press.publish(pressure);
88      rh.publish(humidity);
89  }
90
91  delay(10000);
92  }
93
94  void I2CScan()
95  {
96      byte error, address;
97      int nDevices;
```

```
98
99      Serial.println("--------------------------");
100     Serial.println("-Scanning for I2C Devices-");
101     Serial.println("--------------------------");
102     delay(5000);
103
104     nDevices = 0;
105     for(address = 1; address < 127; address++)
106     {
107         Wire.beginTransmission(address);
108         error = Wire.endTransmission();
109
110         if (error == 0)
111         {
112             Serial.print("I2C device found at address 0x");
113             if (address<16)
114                 Serial.print("0");
115             Serial.print(address,HEX);
116             Serial.println("  !");
117
118             nDevices++;
119         }
120         else if (error==4)
121         {
122             Serial.print("Unknown error at address 0x");
123             if (address<16)
124                 Serial.print("0");
125             Serial.print(address,HEX);
126         }
127     }
128     if (nDevices == 0)
129         Serial.println("No I2C devices found\n");
130     else
131         Serial.println("Done\n");
132
133
134 }
135
136 void printValues()
137 {
138     Serial.print("Temperature = ");
139     Serial.print(bme.readTemperature());
140     Serial.println(" *C");
141
142     Serial.print("Pressure = ");
```

```
143     Serial.print(bme.readPressure() / 100.0F);
144     Serial.println(" hPa");
145 
146     Serial.print("Approx. Altitude = ");
147     Serial.print(bme.readAltitude(SEALEVELPRESSURE_HPA));
148     Serial.println(" m");
149 
150     Serial.print("Humidity = ");
151     Serial.print(bme.readHumidity());
152     Serial.println(" %");
153 
154     Serial.println();
155 }
```

Appendix B

Adafruit IO

Appendix C

Course Outline

Success Coaching with Sue - Weeks 0, 3, 6, and before Grad

C.1 Week 1

Week.Day	Description	Files
1.1	IoT Overview	
	Getting to know Teensy	
	Software Installation - Arduino/Teensyduino	
	This is a breadboard	
	Our first program: Hello World - IoT style	01 blinky.ino
1.2	Basic Circuits - Resistors, Capacitors, Inductors	
	Why Limit Current	
	LEDs	
	Digital vs Analog, Input and Output	
	Introduction to Fritzing	02 helloLED.fzz
	Our first circuit: Hello LED	02 helloLED.ino
	Variables	02 helloLEDvar.ino
1.3	Soldering	
	Schematics	02 helloLED.fzz
	Introduction to 3D Modeling	(revisited)

1.4	Introduction to Serial Monitor Simple input - Buttons Pull Up / Pull Down Controlling an output	 03 button.ino 03 buttonUP.ino 04 buttonLED.ino
1.5	Object Oriented - A simple introduction OneButton Library One button controlling two LEDs Introduction to Arrays	05 oneButton.ino 05 oneButtonLED.ino 05 oneButtonArray.ino 05 oneButtonArray.fzz

C.2 Week 2

Week.Day	Description	Files
2.1	NeoPixel Header Files	06 rainbow.ino 06 rainbowSTR.ino 06 color.h 06 rainbow.fzz
2.2	Encoders Introduction to 3D printing	07 pixelRing.ino 07 pixelRing_on_off.ino
2.3	Serial Communications (GPS or Particle) Serial Peripheral Interface (SPI) File Systems	08 GPS.ino or part.ino 08 GPS.fzz or part.fzz 09 SDmicro.ino 09 SDmicro.fzz
2.4	I2C (inter-IC) interface OLED Displays Integrating what we've learned	10 OLED.ino 10 OLED.fzz 11 GyroRocket.ino 11 GyroRocket.fzz
2.5	Ethernet HTML commands	12 hue.ino 12 hue.fzz 12 hueRND.ino

C.3 Week 3

Week.Day	Description	Files
3.1	WEMO Global Cache Relays	13 wemoFan.ino 14 GC_relay.ino 15 IoTOutlet.ino
3.2	Feedback BME280	16 tempControl.ino
3.3	Printed Circuit Boards Introduction to Voltera	
3.4	Project Integration	
3.5	Project Integration	
4.1	Project Integration	
4.2	Project Integration	

C.4 Week 4

Week.Day	Description	Files
4.1	Hue Lighting Controller - Integration	
4.2	Hue Lighting Controller - Integration	
4.3	Laser Cutting / Laser Engraving Smart Cities - Introduction	
4.4	Particle Argon Getting to know Argon Software Install - Particle Workbench (VSC) As always: Hello World	 Hello_World
4.5	More Particle	

C.5 Week 5 through 7

Week.Day	Description	Files
	MQTT - Adafruit IO	
	Cloud Control of LED	

	NCD Sensors -I2C Gas	
	Smart City Dashboard - Adafruit IO	
	Atlantis - Smart City Intgretion	
	Power Monitoring	
	Vibration Monitoring	
	Intro to Matlab or online FFT	
	Particle Cloud	
	Thingspeak Webhooks	
	RFID	
	Mesh (?)	

Appendix D

Particle Setup and Troubleshooting

D.1 Setting up Particle Argon

These are instructions for how to set up an out-of-the-box fresh-from-the-factory Argon from scratch, without using the mobile apps. I will add the specific instructions for Borons and Xenons later. To complete setup for a Xenon leaf node (no Ethernet Wing), you will need to have an Argon or Boron gateway device already configured and online.

The detailed instructions are part of my Particle Cookbook 20 project (see the "Local Mesh Device Setup" section), but here is an abridged version that assumes that you already know what you're doing for a lot of the steps. I performed this setup on a Mac (OSX 10.14.3, Mojave). Windows or Linux computers may require additional steps to configure drivers, permissions, etc.

Requirements

- A Particle Cloud login account
- Particle CLI version 1.40.0 or higher

- Particle WorkBench, po-util, or other local build toolchain
- DeviceOS 0.9.0 or higher

If you haven't already, make sure that your Particle CLI is updated to the latest version, at least version 1.40.0. You may need to download the newest installer from the Particle web site, or run the command

```
$ particle update-cli
```

(or perhaps run the command npm install -g particle-cli, depending on how you originally installed it.)

If you aren't already, make sure you are logged into your Particle account (particle login).

Unpack your Argon, place it on the breadboard, and connect it to your computer via USB. It should come up blinking blue, indicating "Listening" mode (serial).

If you try to use any Particle CLI commands for your device at this point, you'll find that you can't. You'll have to get the Argon updated to deviceOS 0.9.0 or higher, first. For some reason, I found that I couldn't even use the cli to do a DFU flash, so I ended up using Particle Workbench. I'm guessing that you could also just use the dfu utility from the command line manually, but I did not try that, and I leave that option as an exercise for the reader.

Put your Argon in DFU mode (hold down the RESET and MODE buttons simultaneously, release RESET, then when the status LED flashes yellow, release MODE).

Fire up Workbench and either open an existing project that's set for deviceOS 1.0.0 and an Argon device, or create a new, blank project with those settings. You do not need any particular code, just an empty setup() and loop() are fine.

From the Terminal menu, select Run Build Task..., then choose Particle: Flash application & DeviceOS (local). Wait for Workbench to finish compiling and flashing (you should get a "success" message in the Workbench build

terminal). Your Argon should return to Listening mode (flashing blue). If it gives you a failure the first time, return to DFU mode and run the task again, and it should succeed the second time.

At this point, you should be able to issue a particle serial identify command and it should return your Device ID and deviceOS version. You will need the Device ID in order to claim your device, so keep it handy.

You will now need to update the Argon's NCP (Network CoProcessor), or you won't be able to connect to WiFi. Download the NCP firmware 21 .bin file. Then issue the command

```
$ particle flash --serial argon-ncp-firmware-0.0.5-ota.bin
```

Now, we can configure the Argon for WiFi with particle serial wifi. Follow the prompts to connect to your network. For example:

```
$ particle serial wifi
? Should I scan for nearby Wi-Fi networks? Yes
? Select the Wi-Fi network with which you wish to connect
   your device: myNetwork
? Should I try to auto-detect the wireless security type? Yes
> Detected WPA(PSK/AES,TKIP/TKIP) WPA2(PSK/AES,TKIP/TKIP)
   security
? Wi-Fi Password myWifiPassword
```

Done! Your device should now restart. Your device should connect to WiFi and to the Particle Cloud, and begin breathing cyan.

Now you should be able to claim your device, and rename it:

```
$ particle device add d01c3e6h01bdh3h279c226af
Claiming device d01c3e6h01bdh3h279c226af
Successfully claimed device d01c3e6h01bdh3h279c226af

$ particle device rename d01c3e6h01bdh3h279c226af MyArgon01
Renaming device d01c3e6h01bdh3h279c226af
Successfully renamed device d01c3e6h01bdh3h279c226af to:
   MyArgon01
Lastly, you can create a new Mesh network with your Argon as
   the gateway device:

$ particle mesh create MyMeshNetwork MyArgon01
? Enter a password for the new network [hidden]
```

```
? Confirm the password [hidden]
Done! The device will be registered in the network once it is
    connected to to the cloud.
If all has gone well, you should be done! Your Argon should
   be claimed, named, and ready for action.
```

I'll write up instructions with the differences for a Xenon and Boron when I get the chance. But this outline will probably be a good start for the intrepid who are too impatient to wait for me.

USB Serial Number - Linux Open a command shell and enter the command:

```
lsusb -d 2b04: -v
```

Note that there is a colon after 2b04. This will display verbose information (-v) about all Particle devices (2b04).

Your Device ID is in the iSerial field of the Device Descriptor.

D.2 Setting up Particle Mesh and Xenon

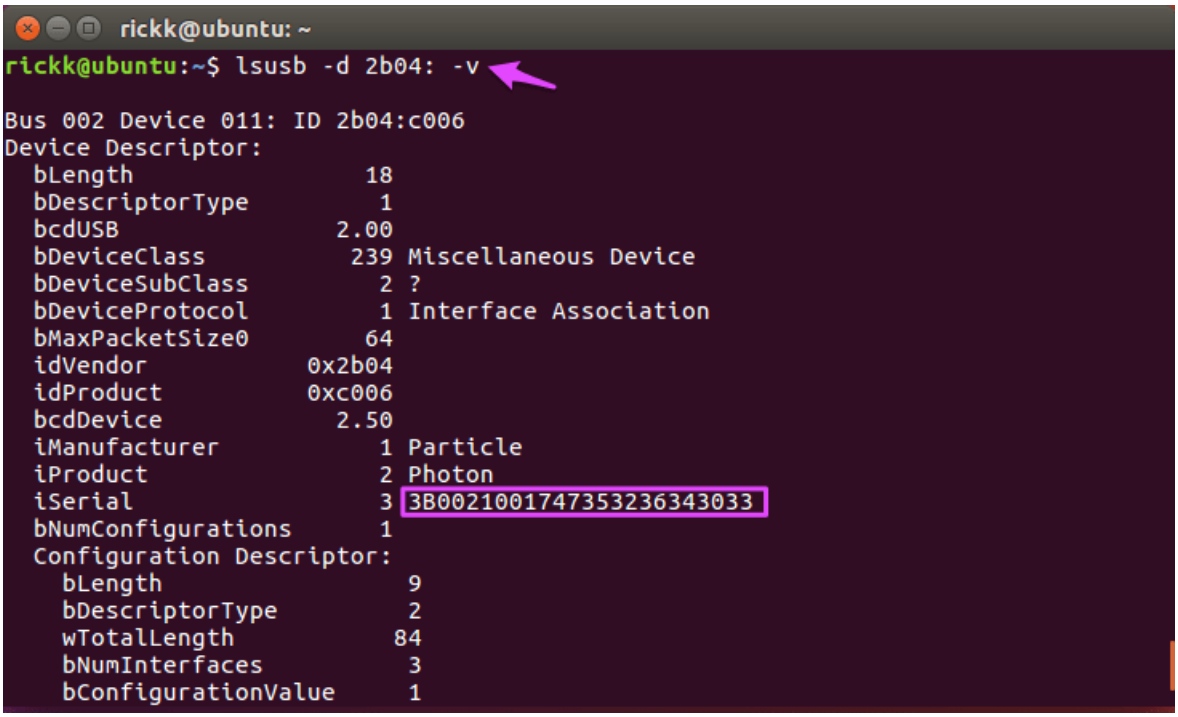
Create a new mesh network. You'll need the device ID of the gateway, the name for your new network, and you'll assign a new password to your mesh network:

```
$ particle mesh create <mesh name> <argon device name>
? Enter a password for the new network [hidden]
? Confirm the password [hidden]
```

where ¡mesh name¿ is the name you want for the mesh, and ¡argon device name¿ is the name of the argon that will serve as the gateway. The device will be registered in the network once it is connected to the cloud.

Put the Xenon in DFU mode (blinking yellow) by holding down MODE. Tap RESET and continue to hold down MODE. The status LED will blink magenta (red and blue at the same time), then yellow. Release when it is blinking yellow.

Figure D.1: USB Serial Number



Update the device. If the device goes out of blinking yellow after the first command, put it back into DFU mode.

```
$ particle update
$ particle flash --usb tinker
```

When the command reports Flash success!, reset the Xenon. It should go back into listening mode (blinking dark blue).

Verify that the update worked:

```
$ particle serial identify
Your device id is e00fce68ffffffc03c6db46a
Your system firmware version is 0.2.1
Save the device ID, you'll need it later.
```

Join the Xenon to your mesh network. You'll need the password for the

network you assigned when you set up your mesh gateway. In this example, we're connecting to the argon3 gateway we set up, above.

```
$ particle mesh add e00fce68ffffffc03c6db46a <argon device
    name>
```

The first parameter is the device ID of the Xenon The second parameter, <argon device name> is the name of the assisting device, typically the gateway (Argon, Boron, or Xenon with Ethernet FeatherWing).

Name your Xenon (optional):

```
particle device rename e00fce68ffffffc03c6db46a <xenon name>
```

Get a list of the devices on your mesh network.

```
particle mesh list
meshtest3
  devices:
    argon3 [e00fce681ffffffffc08949b] (Argon)
    xenon31 [e00fce68ffffffc03c6db46a] (Xenon)
```

Repeat to add more Xenons.

D.3 Argon does not enter DFU mode

```
brian@w1n5t0:~$ particle update-cli
Updating CLI... no plugins to update.
brian@w1n5t0:~$ particle flash --usb tinker

!!! I was unable to detect any devices in DFU mode...

> Your device will blink yellow when in DFU mode.
> If your device is not blinking yellow, please:

1) Press and hold both the RESET/RST and MODE/SETUP buttons
    simultaneously.

2) Release only the RESET/RST button while continuing to hold
     the MODE/SETUP button.

```

```
3) Release the MODE/SETUP button once the device begins to
   blink yellow.

Error writing firmware: No DFU device found
brian@w1n5t0:~$ particle flash --usb tinker
```

D.4 Change WiFi network for Argon

1. Determine the COM port of the Particle Device
   - Windows: Use the Device Manager
   - Mac OS X, in terminal

     ```
     ls /dev/tty.*

     ```

   - Linux:

     ```
     $ dmesg | grep tty

     ```

2. On the Argon long press the Mode button until it turns blue to get the device into Listening Mode
3. using particle-cli

   ```
   particle serial wifi --port <COMx> --verbose
   ```

   where <Comx> is the COM port that the Argon is attached to.
4. followed the setup, the device restarted automatically

D.5 Particle Developers Kit CLI

https://docs.particle.io/reference/developer-tools/cli/

Appendix E

HUE Hub

E.1 Getting HUE hub username

1. Get Chrome extension "Disable Content Security Policy" to allow https:
   - https://chrome.google.com/webstore/detail/disable-content-security/ieelmcmcagommplceebfedjlakkhpden?hl=en
2. Next to the toolbar, press the Content-Security-Header button
3. Find the IP address of the HUE hub
   - You should be able to use AngryIP Scanner on Windows, Mac OS10 and Linux. Download from https://angryip.org/download
4. In your browser: https://¡bridge ip address¿/debug/clip.html
5. In the URL field replace /api/1234/ with /api/newdeveloper and press GET. Your will see the following in Command Response window:

```
[
  {
    "error": {
      "type": 1,
      "address": "/",
      "description": "unauthorized user"
    }
```

```
    }
  ]

```

6. In the URL field type /api
   in the Body field type

```
  {"devicetype":"my_hue_app <name>}

```

   where <name> is what you want to label the device. Press the LINK button on the HUE hub and then within 30 seconds press POST. Your will see the following in Command Response window:

```
  [
    {
      "success":  {
        "username":  "
   MQlZziRO0Wai5MsMHll8xAUAQqw85Qrr8tM37F3T"
      }
    }
  ]

```

7. In URL field type: /api/MQlZziRO0Wai5MsMHll8xAUAQqw85Qrr8tM37F3T/lights and press GET. You will see all the lights on your system.

```
{
  "1": {
    "state": {
      "on": true,
      "bri": 254,
      "hue": 23188,
      "sat": 254,
      "effect": "none",
      "xy": [
        0.2191,
        0.6631
      ],
      "ct": 153,
      "alert": "select",
      "colormode": "hs",
      "mode": "homeautomation",
      "reachable": true
    },
```

```
    "swupdate": {
      "state": "noupdates",
      "lastinstall": "2020-01-20T21:47:57"
    },
    "type": "Extended color light",
    "name": "Hue color lamp 1",
    "modelid": "LCT016",
    "manufacturername": "Philips",
    "productname": "Hue color lamp",
    "capabilities": {
      "certified": true,
      "control": {
        "mindimlevel": 1000,
        "maxlumen": 800,
        "colorgamuttype": "C",
        "colorgamut": [
          [
            0.6915,
            0.3083
          ],
          [
            0.17,
            0.7
          ],
          [
            0.1532,
            0.0475
          ]
        ],
        "ct": {
          "min": 153,
          "max": 500
        }
      },
      "streaming": {
        "renderer": true,
        "proxy": true
      }
    },
    "config": {
      "archetype": "sultanbulb",
      "function": "mixed",
      "direction": "omnidirectional",
      "startup": {
        "mode": "safety",
```

```
        "configured": true
      }
    },
    "uniqueid": "00:17:88:01:03:8f:bd:63-0b",
    "swversion": "1.46.13_r26312",
    "swconfigid": "9DC82D22",
    "productid": "Philips-LCT016-1-A19ECLv5"
  },
  "2": {
    "state": {
      "on": true,
      "bri": 254,
      "hue": 8418,
      "sat": 140,
      "effect": "none",
      "xy": [
        0.4573,
        0.41
      ],
      "ct": 366,
      "alert": "select",
      "colormode": "ct",
      "mode": "homeautomation",
      "reachable": true
    },
    "swupdate": {
      "state": "noupdates",
      "lastinstall": "2020-01-20T21:47:52"
    },
    "type": "Extended color light",
    "name": "Hue color lamp 2",
    "modelid": "LCT016",
    "manufacturername": "Philips",
    "productname": "Hue color lamp",
    "capabilities": {
      "certified": true,
      "control": {
        "mindimlevel": 1000,
        "maxlumen": 800,
        "colorgamuttype": "C",
        "colorgamut": [
          [
            0.6915,
            0.3083
          ],
```

```
            [
              0.17,
              0.7
            ],
            [
              0.1532,
              0.0475
            ]
          ],
          "ct": {
            "min": 153,
            "max": 500
          }
        },
        "streaming": {
          "renderer": true,
          "proxy": true
        }
      },
      "config": {
        "archetype": "sultanbulb",
        "function": "mixed",
        "direction": "omnidirectional",
        "startup": {
          "mode": "safety",
          "configured": true
        }
      },
      "uniqueid": "00:17:88:01:04:2a:f6:2f-0b",
      "swversion": "1.46.13_r26312",
      "swconfigid": "9DC82D22",
      "productid": "Philips-LCT016-1-A19ECLv5"
    }
}

```


8. Finally, in URL: /api/MQlZziRO0Wai5MsMHll8xAUAQqw85Qrr8tM37F3T/lights/2/state. And, try the following three commands, one at a time, in Message Body, pressing PUT to activate.


```
{"on":false}
{"on":true}
{"on":true,"sat":255,"bri":255,"hue":46000}

```

Congratulations, you are good to go.

E.2 Arduino code to control Hue lights from HUB via ethernet

```
/*
 *  Project: Hue Light Test
 *  Description: Test Hue Lights at IOT Classroom
 *  Authors:  Frank Gonzales and Brian Rashap
 *  Date:     20-Jan-2020
 */
#include <SPI.h>
#include <Ethernet.h>

//  Hue constants

const char hueHubIP[] = "172.16.0.2";       // Hue hub IP
const char hueUsername[] = "
   MQlZziROOWai5MsMHll8xAUAQqw85Qrr8tM37F3T";
const int hueHubPort = 80;   // HTTP: 80, HTTPS: 443, HTTP-
   PROXY: 8080

// PIR
int pir = 2;
boolean activated = true;
int bulb = 1;

//  Hue variables
boolean hueOn;  // on/off
int hueBri;     // brightness value
long hueHue;    // hue value
String hueCmd;  // Hue command

unsigned long buffer=0;  //buffer for received data storage
unsigned long addr;

//  Ethernet

byte mac[] = {0x90,0xA2,0xDA,0x00,0x55,0x8D};  // W5100 MAC
   address
IPAddress ip(172,16,0,5);               // Arduino IP
EthernetClient HueClient;
```

```
35
36 void setup()
37 {
38    pinMode(10, OUTPUT);
39   digitalWrite(10, HIGH);
40
41   pinMode(4, OUTPUT);
42   digitalWrite(4, HIGH);
43
44   pinMode(23, OUTPUT);
45   digitalWrite(23, HIGH);
46
47   Serial.begin(9600);
48   Ethernet.begin(mac,ip);
49   delay(2000);
50   Serial.print("LinkStatus: ");
51   Serial.println(Ethernet.linkStatus());
52
53   //pinMode(pir,INPUT_PULLUP);
54   pinMode(pir,INPUT);
55   delay(2000);
56   Serial.println("Ready.");
57 }
58
59 void loop()
60 {
61     Serial.print("Switch Setting: ");
62     Serial.print(digitalRead(pir));
63     Serial.print(" - Bulb#: ");
64     Serial.println(bulb);
65     //Serial.println("Bulb state: ");
66     //Serial.println(getHue(bulb));
67
68     if(digitalRead(pir)==1 && activated == false) {
69       Serial.println(" - activated");
70       delay(500);
71       activated = true;
72       int RNDhue = random(1,65000);
73       String command = "{\"on\":true,\"sat\":255,\"bri
    \":255,\"hue\":";
74       command = command + String(RNDhue) + "}";
75       setHue(bulb,command);
76     }
77
78   if(digitalRead(pir)==0 && activated == true) {
```

```
79        Serial.println(" - DEactivated");
80        delay(500);
81        activated = false;
82        String command = "{\"on\":false}";
83        setHue(bulb,command);
84    }
85    delay(5000);
86 }
87
88 /* setHue() is our main command function, which needs to be
      passed a light number and a
89  * properly formatted command string in JSON format (
      basically a Javascript style array of variables
90  * and values. It then makes a simple HTTP PUT request to the
       Bridge at the IP specified at the start.
91  */
92 boolean setHue(int lightNum,String command)
93 {
94    if (HueClient.connect(hueHubIP, hueHubPort))
95    {
96      //while (HueClient.connected())
97      //{
98        Serial.println("Sending Command to Hue");
99        Serial.println(command);
100       HueClient.print("PUT /api/");
101       HueClient.print(hueUsername);
102       HueClient.print("/lights/");
103       HueClient.print(lightNum);  // hueLight zero based, add
      1
104       //  HueClient.println("/state");
105       HueClient.println("/state HTTP/1.1");
106       HueClient.println("keep-alive");
107       HueClient.print("Host: ");
108       HueClient.println(hueHubIP);
109       HueClient.print("Content-Length: ");
110       HueClient.println(command.length());
111       HueClient.println("Content-Type: text/plain;charset=UTF
     -8");
112       HueClient.println();  // blank line before body
113       HueClient.println(command);  // Hue command
114       Serial.println("From Hue");
115       Serial.println(HueClient.readString()); // To close
     connection
116      HueClient.stop();
117      return true;  // command executed
```

```
118   }
119   else
120     return false;  // command failed
121 }
122
123 /* A helper function in case your logic depends on the
      current state of the light.
124  * This sets a number of global variables which you can check
       to find out if a light is currently on or not
125  * and the hue etc. Not needed just to send out commands
126  */
127 boolean getHue(int lightNum)
128 {
129   if (HueClient.connect(hueHubIP, hueHubPort))
130   {
131     HueClient.print("GET /api/");
132     HueClient.print(hueUsername);
133     HueClient.print("/lights/");
134     HueClient.print(lightNum);
135     HueClient.println(" HTTP/1.1");
136     HueClient.print("Host: ");
137     HueClient.println(hueHubIP);
138     HueClient.println("Content-type: application/json");
139     HueClient.println("keep-alive");
140     HueClient.println();
141     while (HueClient.connected())
142     {
143       if (HueClient.available())
144       {
145         Serial.println();
146         Serial.println(HueClient.readString());
147         Serial.println();
148         HueClient.findUntil("\"on\":", "\0");
149         hueOn = (HueClient.readStringUntil(',') == "true");
    // if light is on, set variable to true
150         Serial.print("Hue Status: ");
151         Serial.println(hueOn);
152
153         HueClient.findUntil("\"bri\":", "\0");
154         hueBri = HueClient.readStringUntil(',').toInt();  //
    set variable to brightness value
155         //Serial.println(hueBri);
156
157         HueClient.findUntil("\"hue\":", "\0");
158         hueHue = HueClient.readStringUntil(',').toInt();  //
```

```
    set variable to hue value
159         //Serial.println(hueHue);
160
161         break;  // not capturing other light attributes yet
162       }
163     }
164     HueClient.stop();
165     return true;  // captured on,bri,hue
166   }
167   else
168     return false;  // error reading on,bri,hue
169 }
```

Appendix F

GITHUB

F.1 Basic GITHUB commands from the CLI

F.1.1 Cloning an existing repository

Use this to get a repository that already exists and pull it to your local machine.

Go the directory where you want to place the repository and:

```
1  git clone <URL of repository>
2
```

F.1.2 Getting the latest updates

To get the latest updates from the repository:

```
1  git pull
2
```

F.1.3 Placing your edits into the repository

To get the latest updates from the respository:

```
git add .
git commit -m "<comment about what you are adding>"
git push

```

Appendix G

Structured Problem Solving

G.1 Problem Solving Flow

- Problem - can you show it in a graph or image
- Segmentation - tree
- Systems Model - drawing
- Model Validation - hypothesis and test
- Solution -
- Solution Validation -
- Systemize

Appendix H

Lineage of Programming Languages

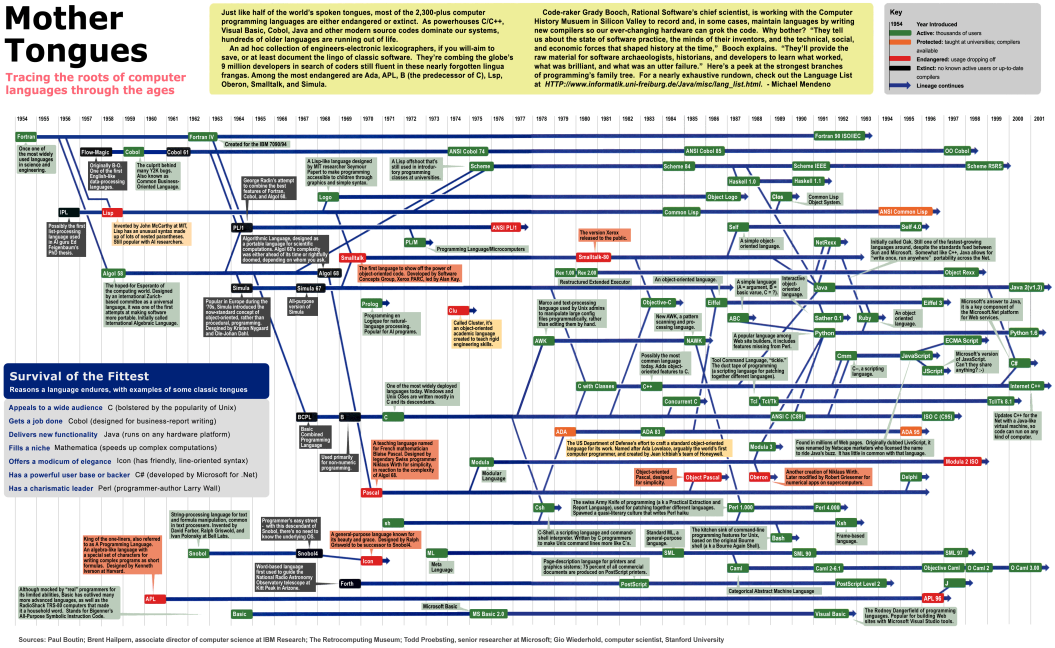
H.1 FORTRAN

Fortran; formerly FORTRAN, derived from Formula Translation[2]) is a general-purpose, compiled imperative programming language that is especially suited to numeric computation and scientific computing.

Originally developed by IBM[3] in the 1950s for scientific and engineering applications, FORTRAN came to dominate this area of programming early on and has been in continuous use for over six decades in computationally intensive areas such as numerical weather prediction, finite element analysis, computational fluid dynamics, computational physics, crystallography and computational chemistry. It is a popular language for high-performance computing[4] and is used for programs that benchmark and rank the world's fastest supercomputers.[5][6]

Fortran encompasses a lineage of versions, each of which evolved to add extensions to the language while usually retaining compatibility with prior versions. Successive versions have added support for structured programming and processing of character-based data (FORTRAN 77), array programming, modular programming and generic programming (Fortran 90), high perfor-

Figure H.1: Program Language Lineage



mance Fortran (Fortran 95), object-oriented programming (Fortran 2003), concurrent programming (Fortran 2008), and native parallel computing capabilities (Coarray Fortran 2008/2018).

Fortran's design was the basis for many other programming languages. Among the better known is BASIC, which is based on FORTRAN II with a number of syntax cleanups, notably better logical structures,[7] and other changes to work more easily in an interactive environment.[8]

H.2 ALGOL

ALGOL 58, originally named IAL, is one of the family of ALGOL computer programming languages. It was an early compromise design soon superseded by ALGOL 60. According to John Backus[2]

"The Zurich ACM-GAMM Conference had two principal motives in proposing the IAL: (a) To provide a means of communicating numerical methods and other procedures between people, and (b) To provide a means of realizing a stated process on a variety of machines..."

ALGOL 58 introduced the fundamental notion of the compound statement, but it was restricted to control flow only, and it was not tied to identifier scope in the way that Algol 60's blocks were.

ALGOL 60 (short for Algorithmic Language 1960) is a member of the ALGOL family of computer programming languages. It followed on from ALGOL 58 which had introduced code blocks and the begin and end pairs for delimiting them. ALGOL 60 was the first language implementing nested function definitions with lexical scope. It gave rise to many other programming languages, including CPL, Simula, BCPL, B, Pascal, and C.

Niklaus Wirth based his own ALGOL W on ALGOL 60 before moving to develop Pascal. Algol-W was intended to be the next generation ALGOL but the ALGOL 68 committee decided on a design that was more complex and advanced rather than a cleaned simplified ALGOL 60. The official ALGOL versions are named after the year they were first published. Algol 68 is substantially different from Algol 60 and was criticised partially for being so, so that in general "Algol" refers to dialects of Algol 60.

H.3 CPL

CPL (Combined Programming Language) is a multi-paradigm programming language, that was developed in the early 1960s. It is an early ancestor of the C language via the BCPL and B languages.

H.4 BCPL

BCPL ("Basic Combined Programming Language") is a procedural, imperative, and structured computer programming language. Originally intended

for writing compilers for other languages, BCPL is no longer in common use. However, its influence is still felt because a stripped down and syntactically changed version of BCPL, called B, was the language on which the C programming language was based. BCPL introduced several features of many modern programming languages, including using curly braces to delimit code blocks.[3] BCPL was first implemented by Martin Richards of the University of Cambridge in 1967.

H.5 B

B is a programming language developed at Bell Labs circa 1969. It is the work of Ken Thompson with Dennis Ritchie.

B was derived from BCPL, and its name may be a contraction of BCPL. Thompson's coworker Dennis Ritchie speculated that the name might be based on Bon, an earlier, but unrelated, programming language that Thompson designed for use on Multics.

B was designed for recursive, non-numeric, machine-independent applications, such as system and language software.[3] It was a typeless language, with the only data type being the underlying machine's natural memory word format, whatever that might be. Depending on the context, the word was treated either as an integer or a memory address.

As machines with ASCII processing became common, notably the DEC PDP-11 that arrived at Bell, support for character data stuffed in memory words became important. The typeless nature of the language was seen as a disadvantage, which led Thompson and Ritchie to develop an expanded version of the language supporting new internal and user-defined types, which became the C programming language.

H.6 C

C, as in the letter c) is a general-purpose, procedural computer programming language supporting structured programming, lexical variable scope, and recursion, while a static type system prevents unintended operations. By design, C provides constructs that map efficiently to typical machine instructions and has found lasting use in applications previously coded in assembly language. Such applications include operating systems and various application software for computers, from supercomputers to embedded systems.

C was originally developed at Bell Labs by Dennis Ritchie between 1972 and 1973 to make utilities running on Unix. Later, it was applied to reimplementing the kernel of the Unix operating system.[6] During the 1980s, C gradually gained popularity. It has become one of the most widely used programming languages,[7][8] with C compilers from various vendors available for the majority of existing computer architectures and operating systems. C has been standardized by the ANSI since 1989 (see ANSI C) and by the International Organization for Standardization.

C is an imperative procedural language. It was designed to be compiled using a relatively straightforward compiler to provide low-level access to memory and language constructs that map efficiently to machine instructions, all with minimal runtime support. Despite its low-level capabilities, the language was designed to encourage cross-platform programming. A standards-compliant C program written with portability in mind can be compiled for a wide variety of computer platforms and operating systems with few changes to its source code. The language is available on various platforms, from embedded microcontrollers to supercomputers.

H.7 C++

C++ is a general-purpose programming language created by Bjarne Stroustrup as an extension of the C programming language, or "C with Classes". The language has expanded significantly over time, and modern C++ has

object-oriented, generic, and functional features in addition to facilities for low-level memory manipulation. It is almost always implemented as a compiled language, and many vendors provide C++ compilers, including the Free Software Foundation, LLVM, Microsoft, Intel, Oracle, and IBM, so it is available on many platforms.[6]

C++ was designed with a bias toward system programming and embedded, resource-constrained software and large systems, with performance, efficiency¸ and flexibility of use as its design highlights.[7] C++ has also been found useful in many other contexts, with key strengths being software infrastructure and resource-constrained applications,[7] including desktop applications, servers (e.g. e-commerce, Web search, or SQL servers), and performance-critical applications (e.g. telephone switches or space probes).[8]

C++ is standardized by the International Organization for Standardization (ISO), with the latest standard version ratified and published by ISO in December 2017 as ISO/IEC 14882:2017 (informally known as C++17).[9] The C++ programming language was initially standardized in 1998 as ISO/IEC 14882:1998, which was then amended by the C++03, C++11 and C++14 standards. The current C++17 standard supersedes these with new features and an enlarged standard library. Before the initial standardization in 1998, C++ was developed by Danish computer scientist Bjarne Stroustrup at Bell Labs since 1979 as an extension of the C language; he wanted an efficient and flexible language similar to C that also provided high-level features for program organization.[10] C++20 is the next planned standard, keeping with the current trend of a new version every three years.[11]

H.8 Python

Python is an interpreted, high-level, general-purpose programming language. Created by Guido van Rossum and first released in 1991, Python's design philosophy emphasizes code readability with its notable use of significant whitespace. Its language constructs and object-oriented approach aim to help programmers write clear, logical code for small and large-scale projects.[27]

Python is dynamically typed and garbage-collected. It supports multiple pro-

gramming paradigms, including procedural, object-oriented, and functional programming. Python is often described as a "batteries included" language due to its comprehensive standard library.[28]

Python was conceived in the late 1980s as a successor to the ABC language. Python 2.0, released in 2000, introduced features like list comprehensions and a garbage collection system capable of collecting reference cycles. Python 3.0, released in 2008, was a major revision of the language that is not completely backward-compatible, and much Python 2 code does not run unmodified on Python 3.

The Python 2 language, i.e. Python 2.7.x, is "sunsetting" on January 1, 2020 (after extension; first planned for 2015), and the Python team of volunteers will not fix security issues, or improve it in other ways after that date.[29][30] With the end-of-life, only Python 3.5.x and later will be supported.

Python interpreters are available for many operating systems. A global community of programmers develops and maintains CPython, an open source[31] reference implementation. A non-profit organization, the Python Software Foundation, manages and directs resources for Python and CPython development.

H.9 PHP

PHP: Hypertext Preprocessor (or simply PHP) is a general-purpose programming language originally designed for web development. It was originally created by Rasmus Lerdorf in 1994;[6] the PHP reference implementation is now produced by The PHP Group.[7] PHP originally stood for Personal Home Page,[6] but it now stands for the recursive initialism PHP: Hypertext Preprocessor.[8]

PHP code may be executed with a command line interface (CLI), embedded into HTML code, or used in combination with various web template systems, web content management systems, and web frameworks. PHP code is usually processed by a PHP interpreter implemented as a module in a web server or as a Common Gateway Interface (CGI) executable. The web server outputs

the results of the interpreted and executed PHP code, which may be any type of data, such as generated HTML code or binary image data. PHP can be used for many programming tasks outside of the web context, such as standalone graphical applications[9] and robotic drone control.[10]

The standard PHP interpreter, powered by the Zend Engine, is free software released under the PHP License. PHP has been widely ported and can be deployed on most web servers on almost every operating system and platform, free of charge.[11]

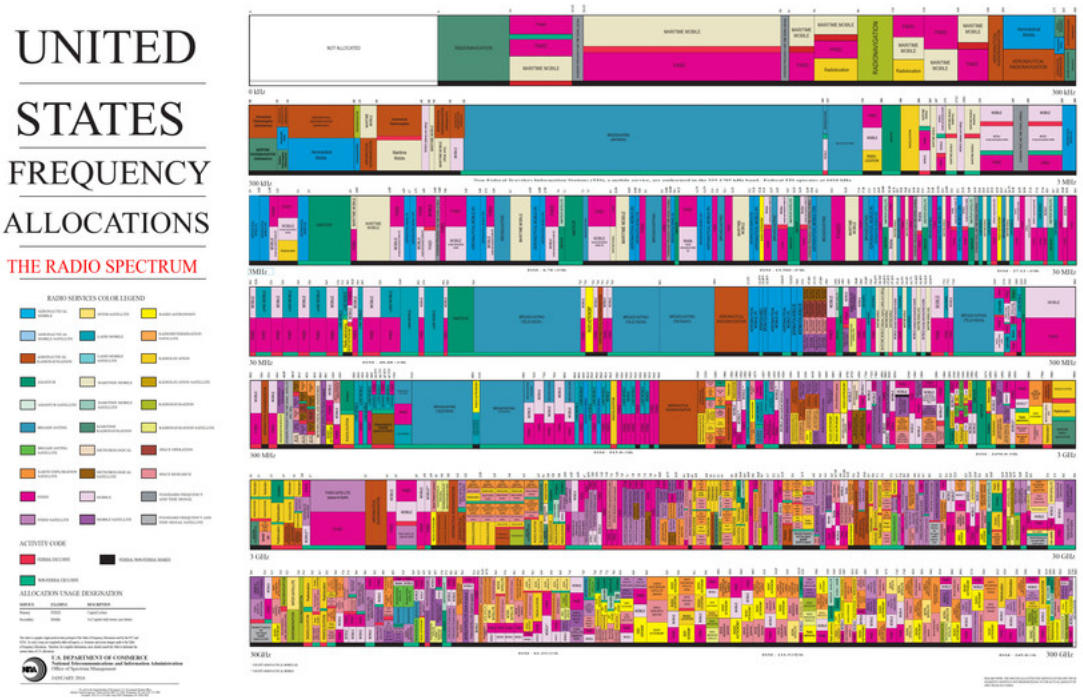
The PHP language evolved without a written formal specification or standard until 2014, with the original implementation acting as the de facto standard which other implementations aimed to follow. Since 2014, work has gone on to create a formal PHP specification.[12]

As of September 2019, over 60% of sites on the web using PHP are still on discontinued/"EOLed"[13] version 5.6 or older;[14] versions prior to 7.1 are no longer officially supported by The PHP Development Team,[15] but security support is provided by third parties, such as Debian.[16]

Appendix I

FCC Frequency Allocation

Figure I.1: FCC Frequency Allocation



Appendix J

Cool little table that I don't want to forget how to do

Classification of the criticel point $(0,0)$ of $x' = Ax, \lvert\mathbf{A}\rvert \neq 0$.		
Types	Type of Critical Point	Stability
1. Real unequal eigenvalues of same sign		
• $\lambda_1 > \lambda_2 > 0$	Improper node/node	Unstable
• $\lambda_1 < \lambda_2 < 0$	Improper node/node	Asym. stable
2. Real unequal eigenvalues of opposite sign		
• $\lambda_2 < 0 > \lambda_1$	Saddle point	Unstable
3. Equal eigenvalues		
Subtype 1: Two Independent vectors		
• $\lambda_1 = \lambda_2 > 0$	Proper node	Unstable
• $\lambda_1 = \lambda_2 < 0$	Proper node	Asym. stable