Learning IR Remote Control Kit

Combine your TV/DVD/air-con/whatever into one remote that you build!

2017

ELECTRICALLY BASED ENGINEERING

STUDENT SOCIETY

Table of Contents

EBESS Kit
Learning IR Remote
2017
Rev1.0

Introduction

What is this?
This kit is a learning IR (infra-red) remote. This means that it is like your TV remote in that it can talk to various devices by emitting special IR signals. It is alearning remote because it has the ability to learn the commands from almost any of your remote controls at home. You can combine your air-conditioning, TV, and whatever else into one remote.

Why is this a kit?
I made this kit because I really wanted engineering students to see that with electrical engineering, you gain the skills to be able to create whatever you can imagine. I want this to give other people ideas to make their own projects to not only improve their skills, but to raise everyone's skills at the same time. Who knows, you could make a kit that other students could be building one day!

I don't know anything about electronics
Don't worry, this kit is meant to be straightforward enough so that even if you haven't had your first engineering class yet, you should be able to build this. All you really need to do is solder the components, as the code and instructions are provided in this document. If you don't know how to solder, or would like some help, we'll be holding sessions to help you out. Just talk to one of the EBESS executives to find out more. Once you can solder, you'll have this sweet remote control up and running.

I'm okay at electronics
Great, you'll get some good experience out of this and learn a bit along the way.

I'm already super awesome at electronics
Even if you're a later year student, this is still a good kit for you. There is no such thing as too much practical experience in electronics, and there is always something new to learn. This might give you design ideas for your own projects, or you could learn some new theory. Either way, you'll get something out of this.

What can this remote learn?
From testing, this remote can learn signals for remotes from TVs, air-conditioning, DVD players, digital set-top boxes, Foxtel and Apple TV (the generic Apple remote). This is by no means an exhaustive list; if you have a remote control that uses IR, it is very likely that this remote will be able to learn the signal. This is because fortunately, almost all remote controls operate in a very similar fashion.

Assembly Instructions

What's in the box?

First of all, let's look at the parts that you've got.

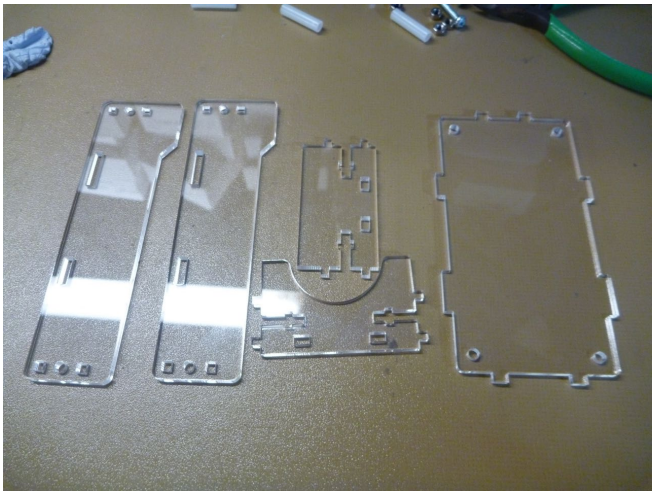


Figure 1: Acrylic case parts (2x sides, 1x rear, 1x front, 1x base)

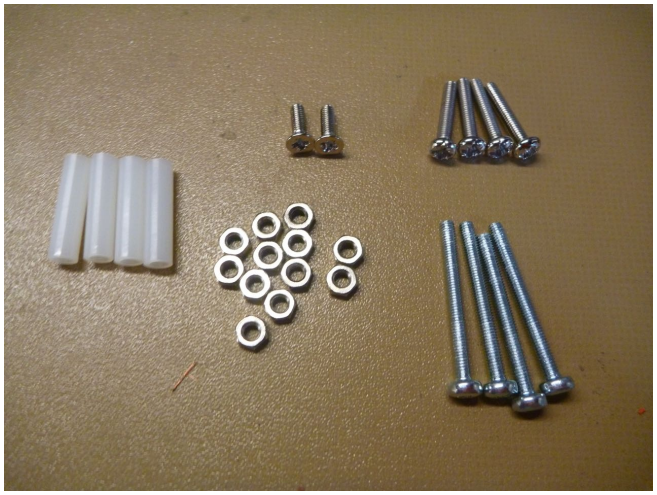


Figure 2: Securing parts (4x20mm nylon spacers, 2x6mm countersunk M3 bolts, 4x16mm M3 bolts, 4x30mm M3 bolts, 12x M3 hex nuts)

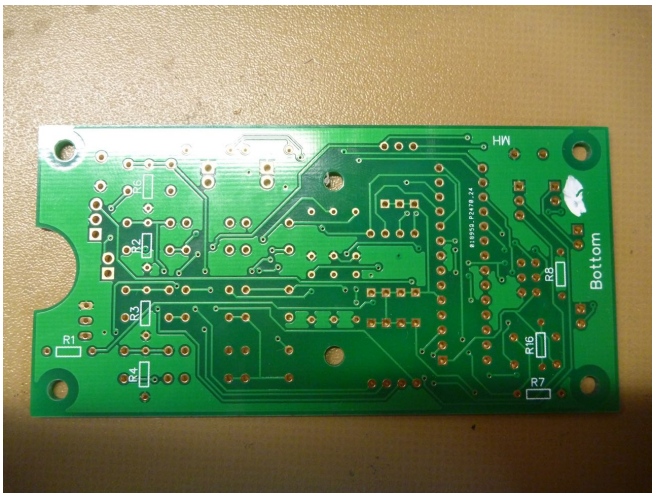


Figure 3: Top side of PCB (printed circuit board)

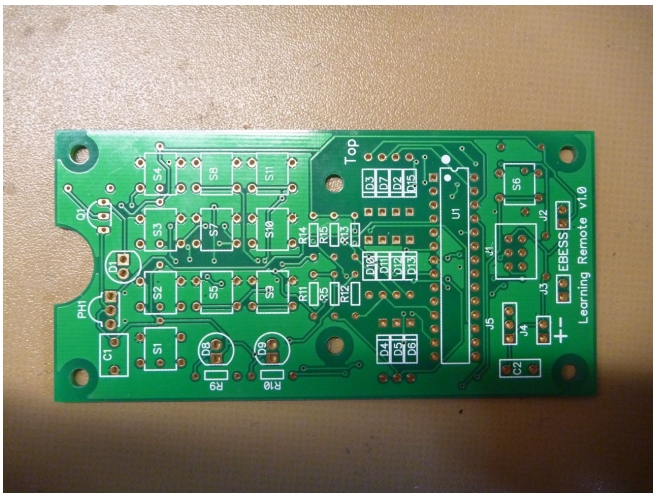


Figure 4: Bottom side of PCB

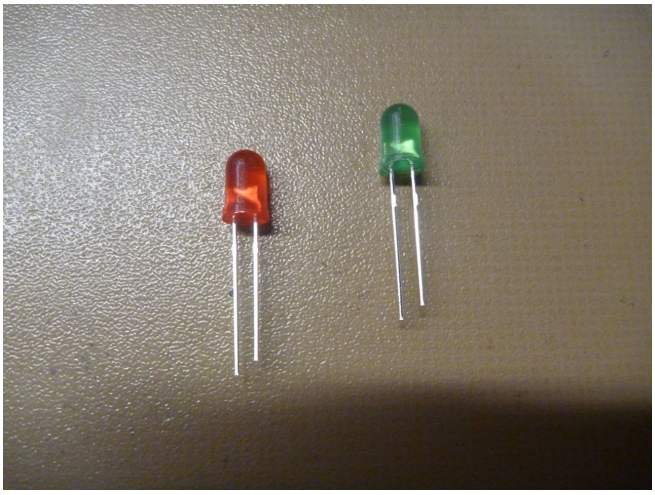


Figure 5: 1x red LED, 1x green LED (both 5mm)

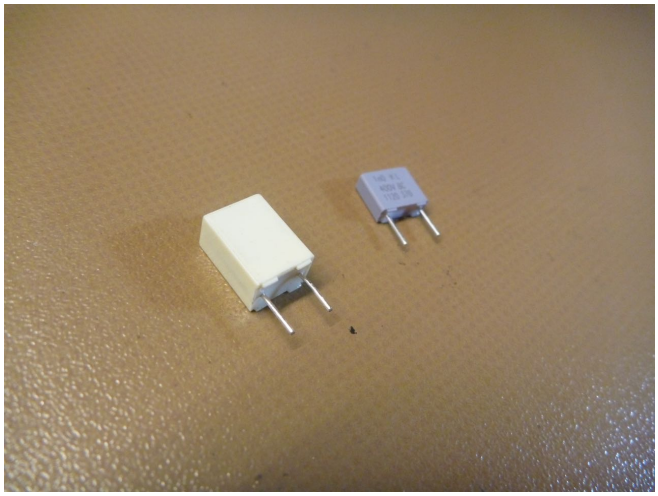


Figure 6: 1x 1uF capacitor (may also be blue or gray), 1x 1nF capacitor

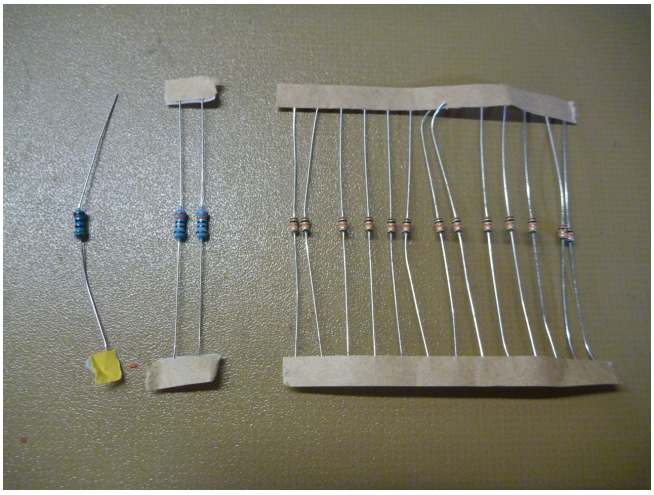


Figure 7: 1x 150Ω resistor, 2x 3.3kΩ resistors, 13x 10kΩ resistors

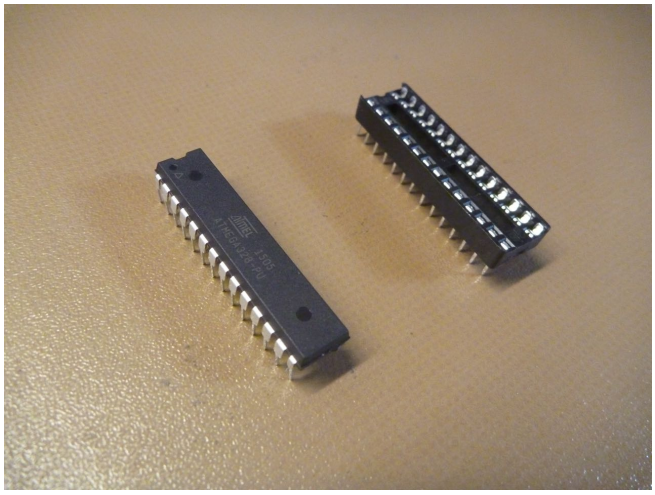


Figure 8: 1x Atmega328 microcontroller, 1x 28 pin DIP (dual inline package) socket

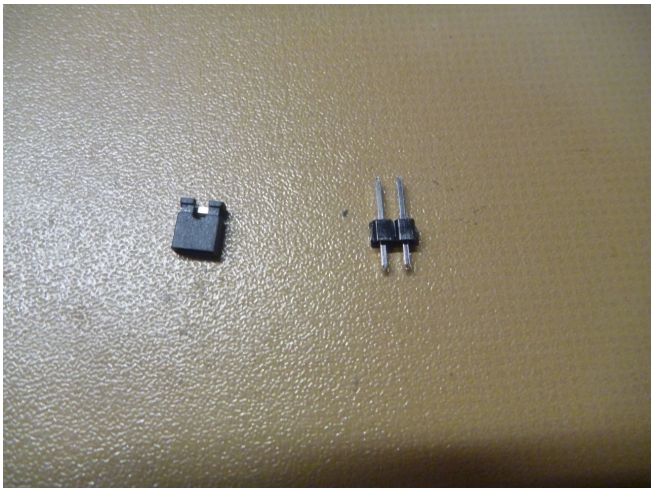


Figure 9: 1x jumper shunt, 1x pair of header pins

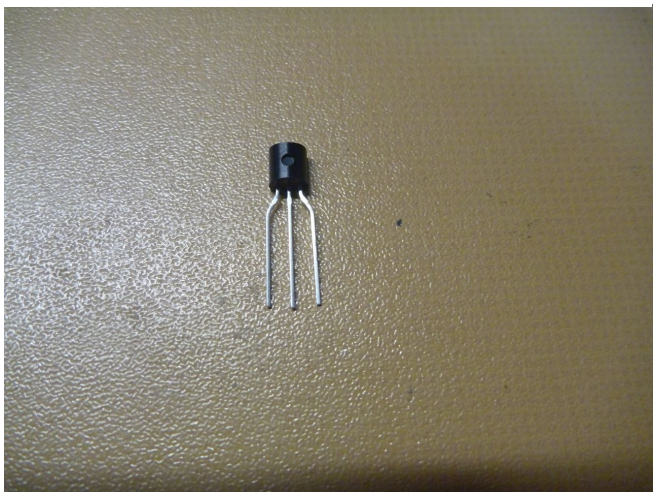


Figure 10: 1x 2N3904 NPN BJT (bipolar junction transistor)

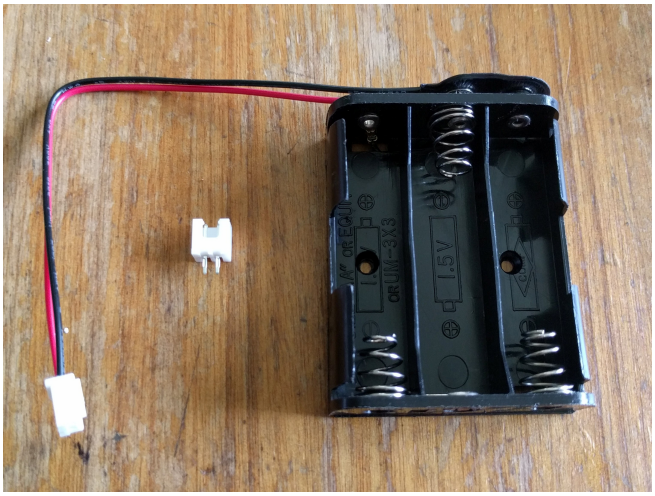


Figure 11: 1x 3AA battery pack with connectors

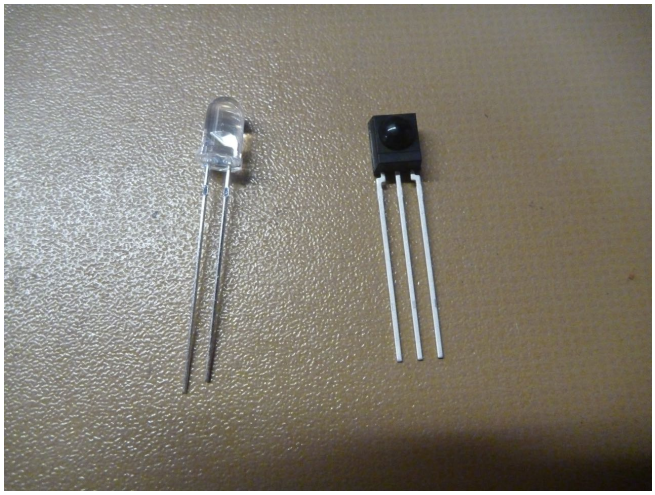


Figure 12: 1x IR emitter, 1x IR demodulator

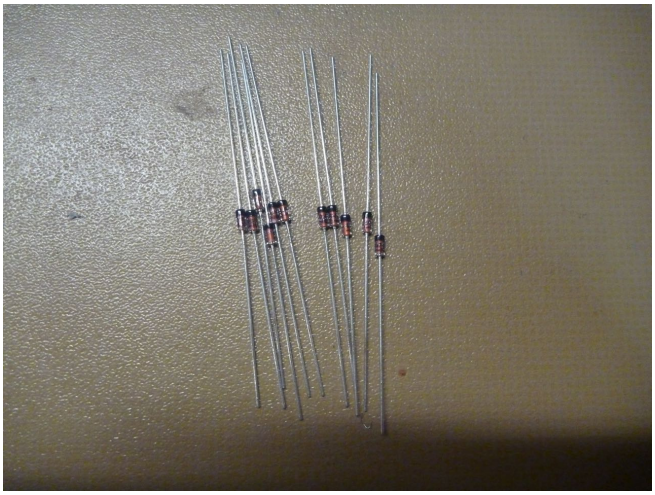


Figure 13: 11x 1N4148 diodes

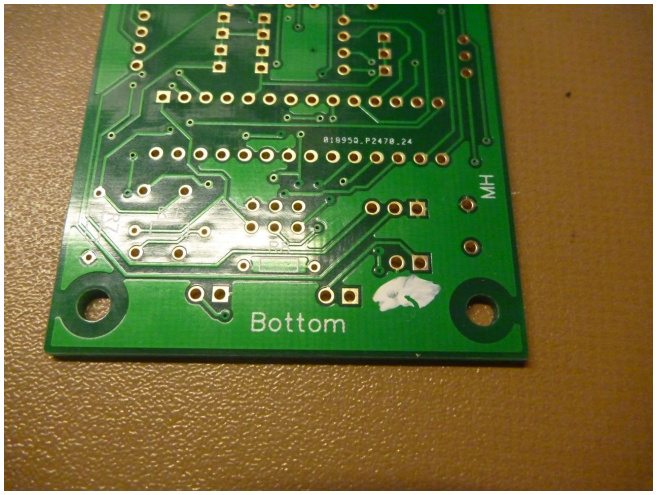
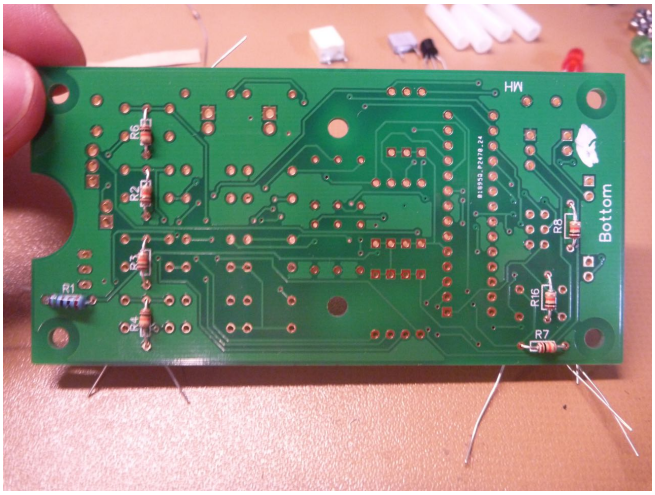


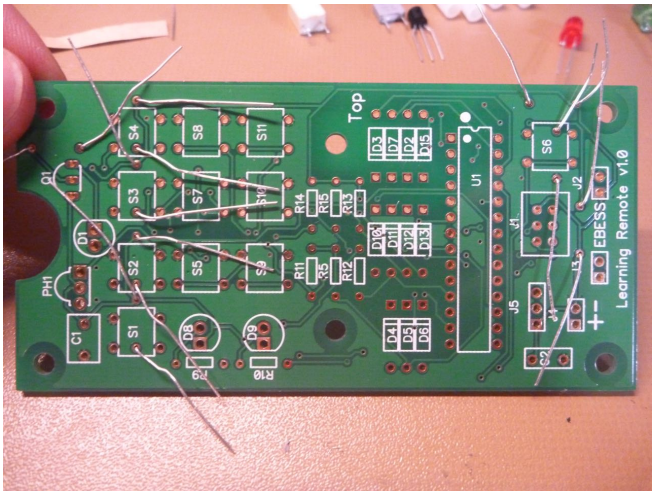
Figure 14: Close up of silkscreen error on PCB

Soldering

Now that we've got all of our parts, we can begin to assemble it. A photo guide is laid out below, but it is only a recommendation. You can of course solder components in any order that you like, but the order is set up conveniently in the photo guide. If you choose to solder this your own way, my only suggestion is to do all the components on the bottom first. This is because some of the resistors come up underneath the buttons, which will be very hard to solder if you do the buttons first.

Let's get into it and build this thing!





Step 1: Insert the resistors on the bottom side of the PCB

Above, we've got the bottom side of the PCB. R1 in the bottom left corner is the 150Ω resistor, and the other seven resistors here are 10kΩ. We just need to simply solder them up on the other side. Don't be too concerned if your solder from the top side doesn't show up on the bottom side; the holes for the resistors are quite narrow and the solder may have difficulty getting through.

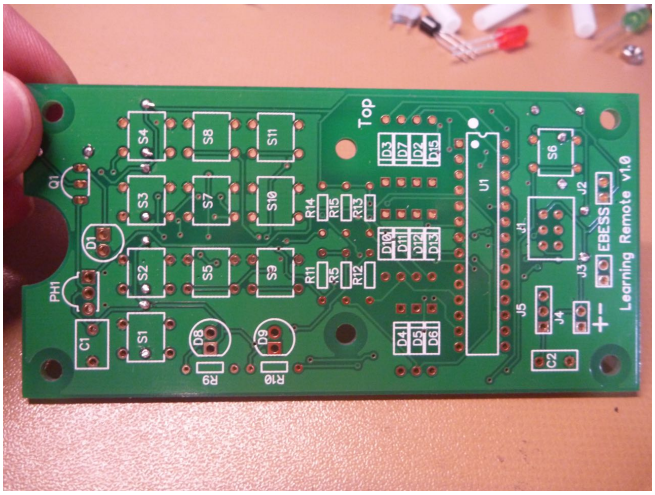
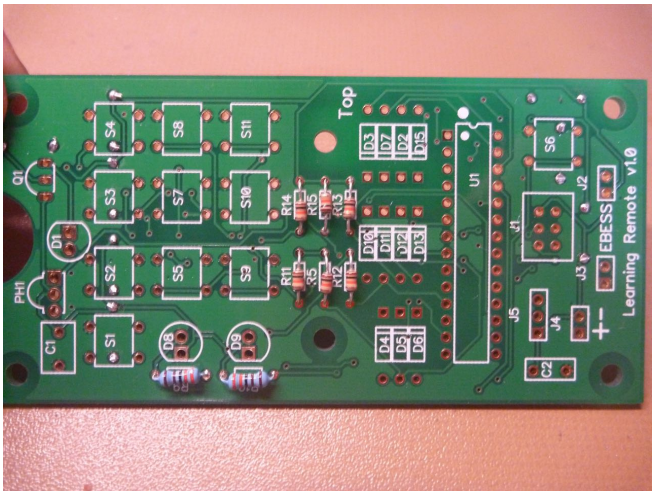


Figure 15: Bottom resistors soldered and their legs snipped on the top side



Step 2: Insert the resistors on the top side

In step 2, we finish off the resistors on the top side by inserting the two 3.3kΩ resistors in R9 and R10 down below, and fill up the other six resistor spots with the remaining 10kΩ resistors (R5, R11-R15). Snip the legs when they're soldered. Once that's all done, it's time to move on to the diodes.

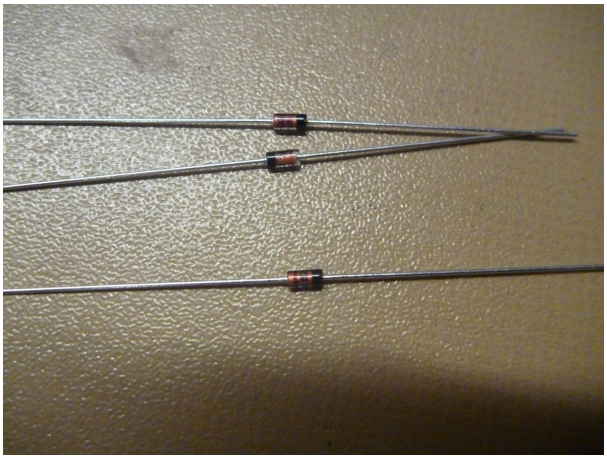
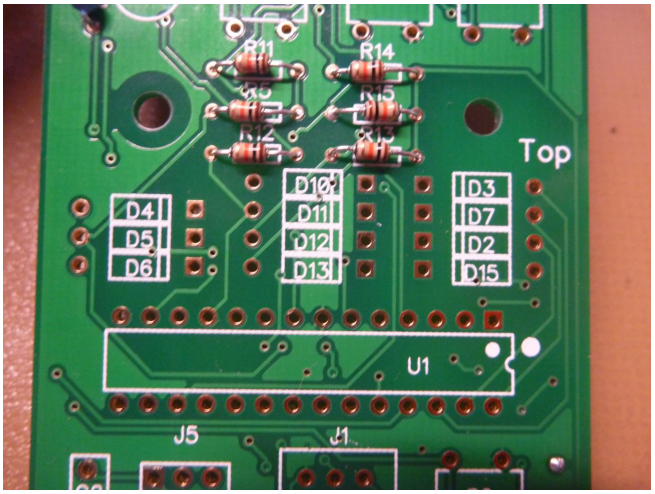
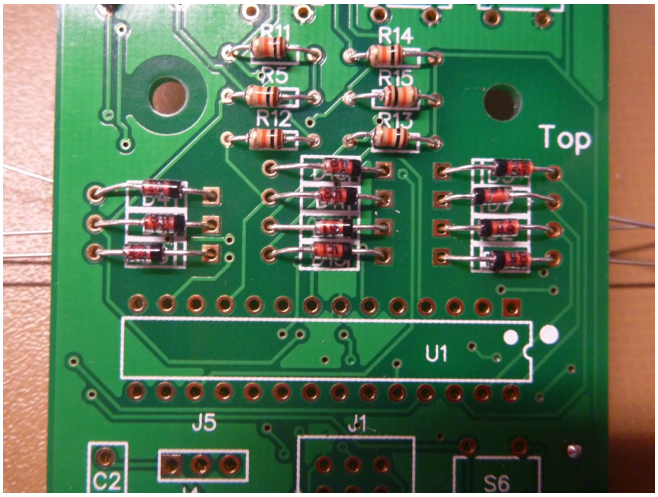


Figure 16: The diodes

You will need to pay careful attention to the orientation of the diodes here. As shown in the figure to the left, you will notice a small black strip on one of the ends of the diodes. This black line has to be aligned with the small strip on the diode silkscreen, shown on the figures just over the page. If reversed, the button assigned to that diode likely won't work.

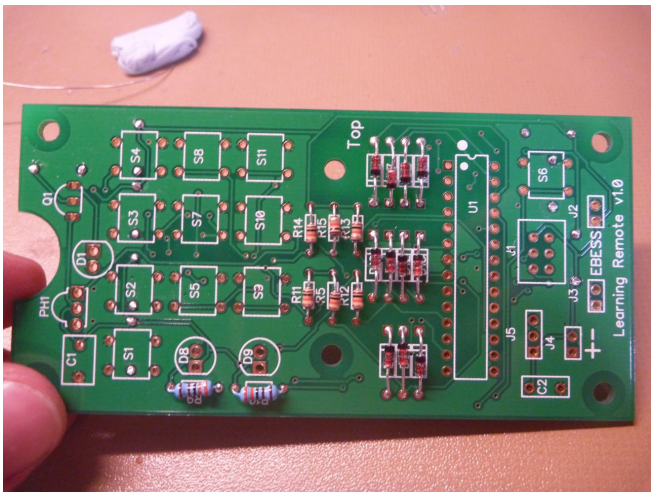


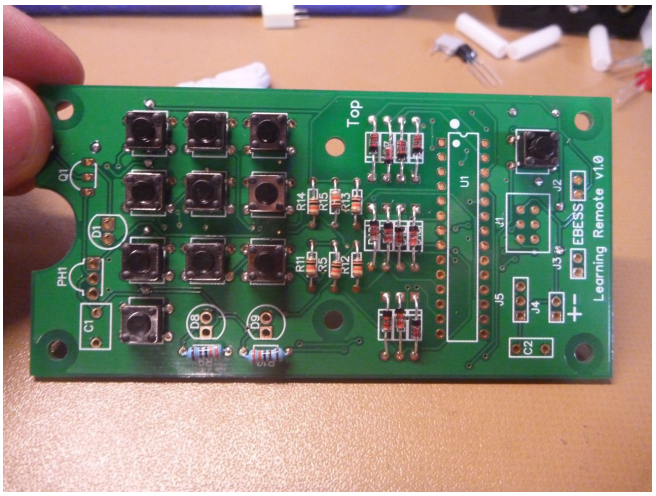


Step 3: Align the diodes to their footprints correctly (note: they're not all the same!)

In the photos for step 3, you'll notice that the previously mentioned black bars on the diodes have been aligned to footprints in the appropriate orientations. The left and centre column of diodes have the bar on the right, and the right column of diodes has the black bar on the left. Make sure you have this the right way around! Once again, snip the legs after soldering.

Now we're going to chuck in all the buttons and solder them up. They will only fit in one orientation, which might involve a minute stretching of the legs (lengthways, not widthways). They populate the spots S1-S11; don't forget the lone button on the other end of the board!



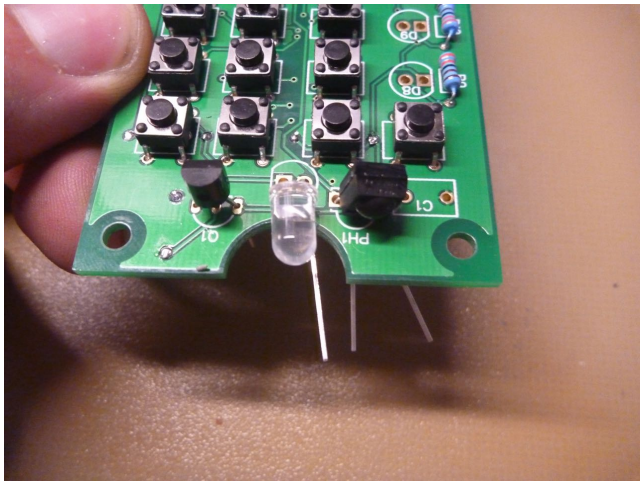


Step 4: Insert the buttons

This step is pretty straightforward. As per the picture, insert all the buttons on top and solder them up. No snipping necessary. We're almost done with soldering!

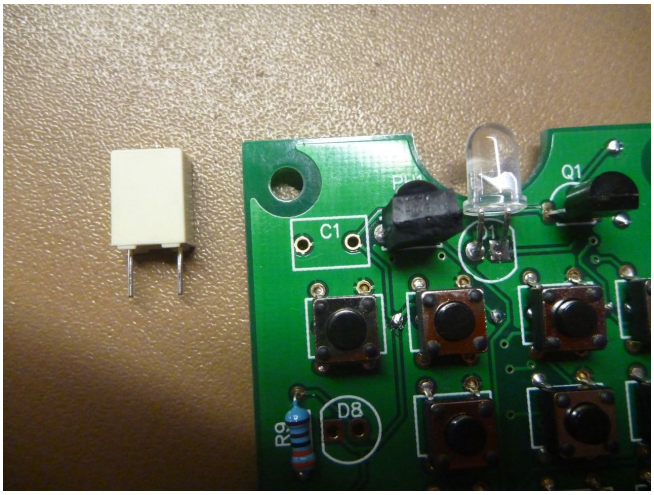
Next up we'll be soldering the key communication components: the IR emitter, the IR demodulator, and the NPN BJT used to drive the IR emitter. As shown over the page, we want to bend the IR emitter such that it points forward. Otherwise you'll have to point your remote very awkwardly to make it work!

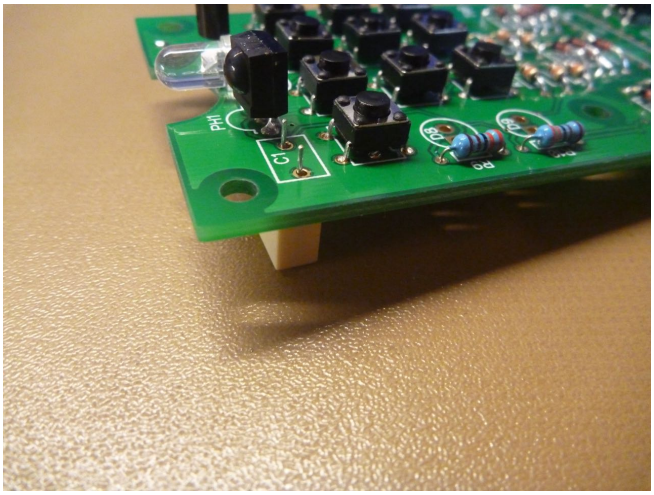
When inserting these three components, pay attention to the footprints for each component, this will tell you how to orient them correctly. The BJT (Q1), and the IR demodulator (PH1) are fairly obvious, but it's a bit trickier for the IR emitter. Notice on its footprint (D1) that there is a noticeable flat side. Now, if you grab your IR emitter, you will want to fill around the rim, just before the legs come out, and you should feel that there is a part of the rim that stops being rounded and flattens out. You want to align this with the flat edge on the footprint of D1. As with every other component, if you don't orient this correctly, your device will not work.



Step 5: Insert transistor, IR emitter and IR demodulator and solder them

Snip the legs when you've soldered these components in. Now we're going to solder the 1uF capacitor in C1.

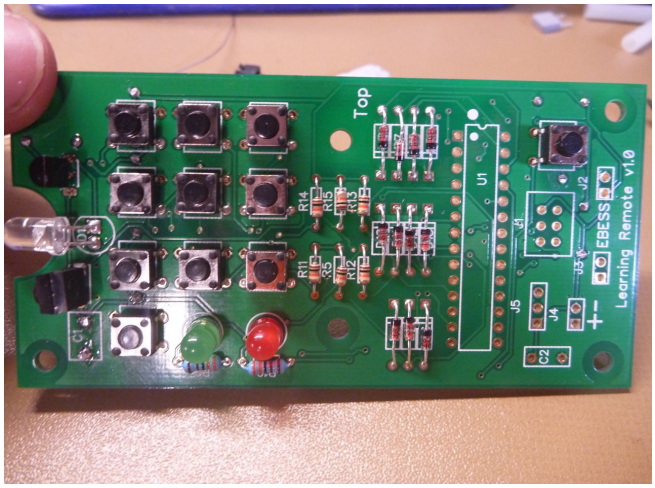




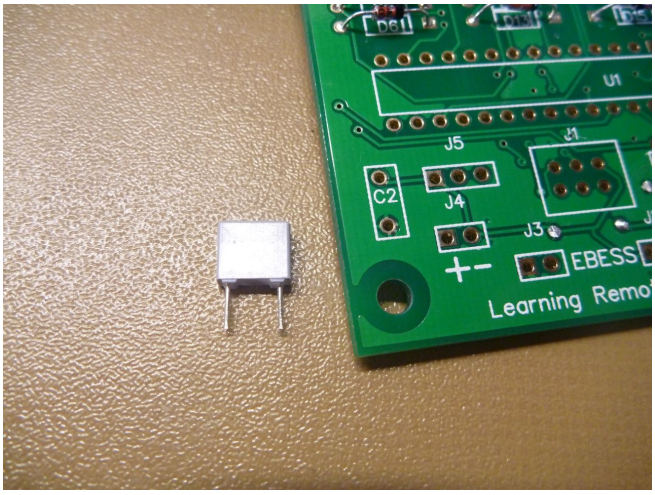
Step 6: 1uF capacitor, inserted upside down (intentionally)

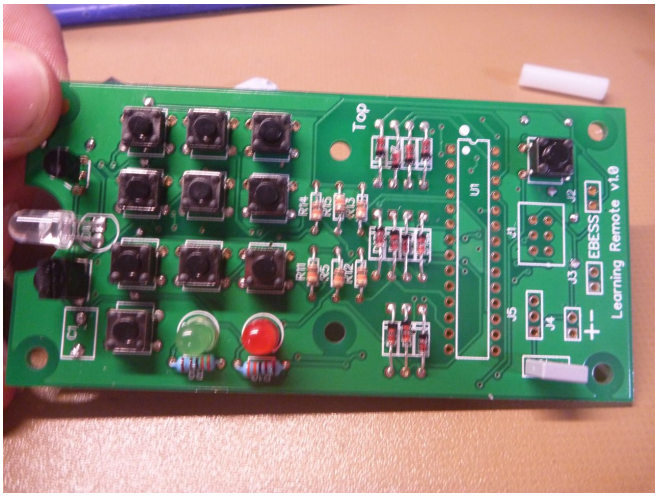
You'll notice that I've inserted the capacitor upside down, that is, on the opposite side of the PCB compared to what its footprint says. I did this because I found the capacitor to be a bit awkward for the user's finger when they are pressing the button next to it. The capacitor has no specific orientation, and placing it on either side of the PCB will have no effect on functionality, so do as you please.

As shown to the right, the green LED goes in the D8 footprint, and the red LED goes in the D9 footprint. Be careful here, as just like with the IR emitter, you need to find the flat side on the rim of each LED and then line that up with the respective flat edge on its footprint. The LEDs have to be the right way around.



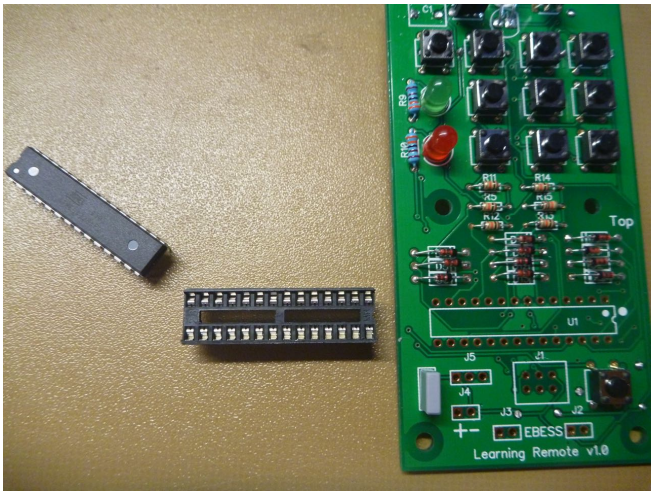
Step 7: Insert and solder the red and green LEDs

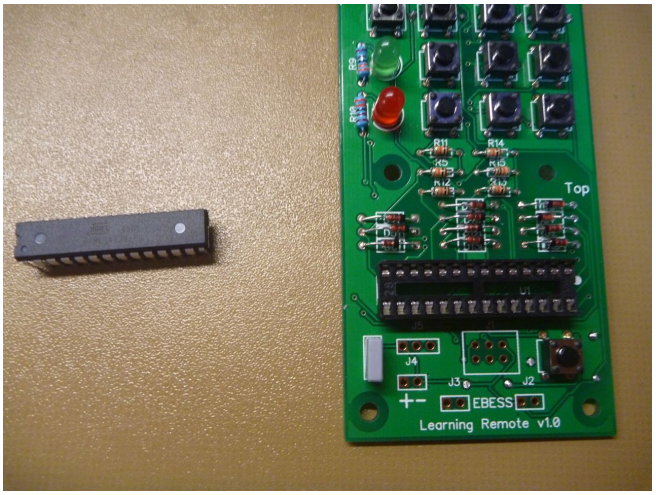




Step 8: Insert and solder the 1nF capacitor, in C2

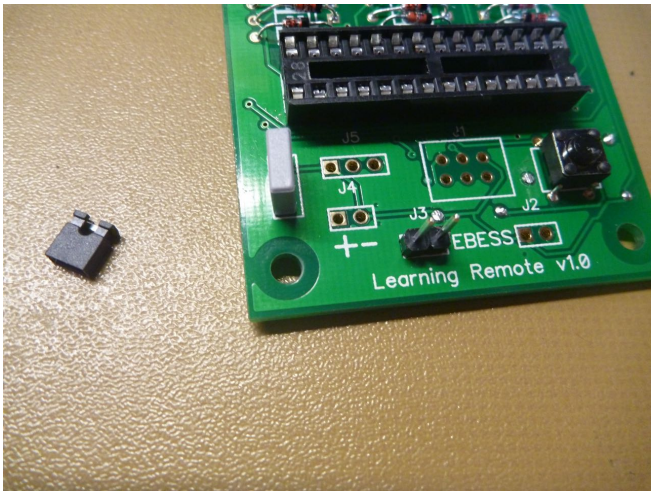
C2 is an easy job. Just chuck the 1nF capacitor in, solder both legs, and you're done. I put it the right way around this time, when compared to C1, because it had no real effect on the user. Try to make it less crooked than I did!





Step 9: Insert the 28 pin DIP socket, but not the microcontroller!

The above step involves soldering the socket. It's a very straightforward job, but I want the emphasis to be that we ONLY solder the socket in, leave the Atmega328 (microcontroller) out of it completely. We'll get to it in a few steps, but now isn't the time. Only two things left to solder now.

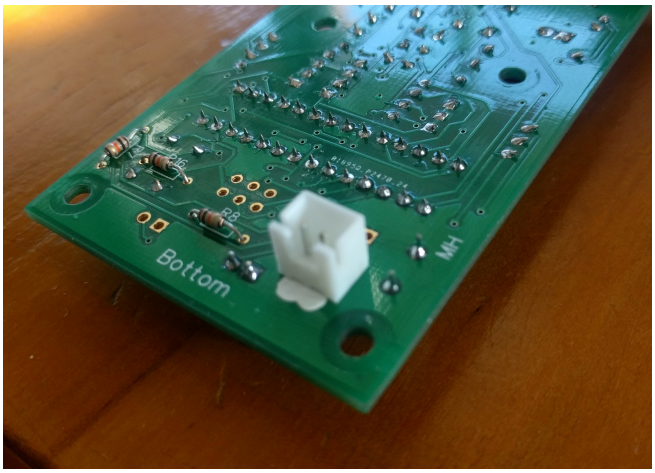


All we're doing here is soldering the pair of header pins into J3, and leaving the jumper shunt out of it. Once we've soldered the battery pack in, the jumper shunt will be responsible for allowing the circuit to be on or off. It remains in the design for debugging purposes (current measurement), so as the user, all you need to know is that the jumper shunt has to be on those pins for the circuit to be on.

Step 10: Solder the pair of header pins

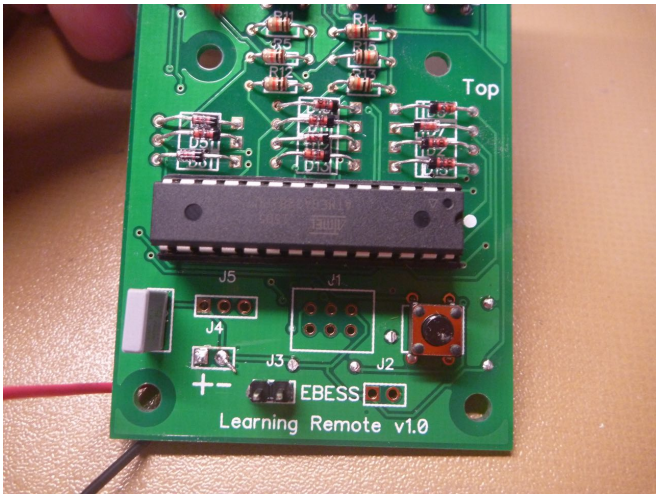
Last, but absolutely not least, we are going to solder the battery pack connector in. Insert the connector socket on the bottom of the PCB into the J4 pins. Make sure to orient it the right way, as in the photo below. The opening should be facing the back of the remote. Once this is done, you're right to connect the battery pack straight to it.

Please ignore any future pictures that have the battery packs wires soldered directly to the PCB.



Step 11: Insert and solder the battery pack connector socket

And that's it for soldering! Now we just have to put the microcontroller and batteries in, then some nuts and bolts for the mounting, and we'll be done! Good work so far, it's been a long slog. What about J1, J2 and J5? Those aren't necessary, they are for debugging or programming purposes, which as the end user you don't need. If you would like to interface with them, you can buy some header pins in and mess around however you would like.

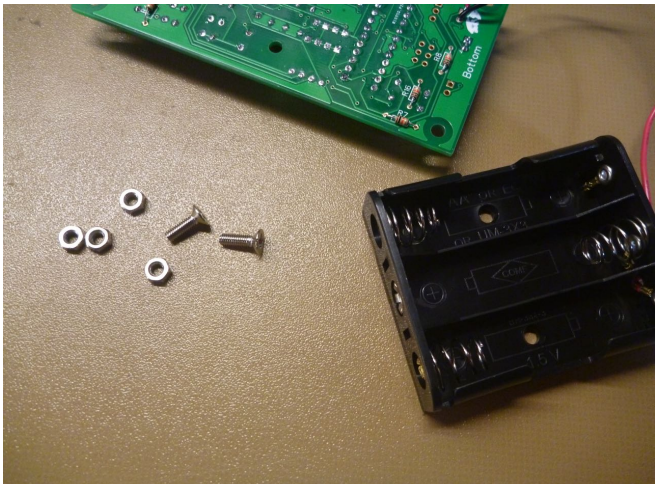


In this step, it is very important to be careful. The first thing to notice is that on top of your Atmega328, there are three circles. In the picture to the left, let's look at the smaller one at the top right of the microcontroller. This HAS to line up with the white circle on the PCB that you'll notice to the right of the socket and microcontroller.

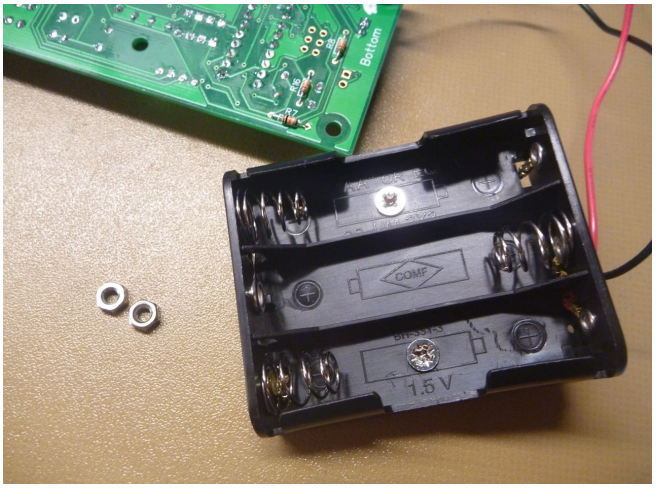
Step 12: Put the microcontroller in

Screwing the parts together

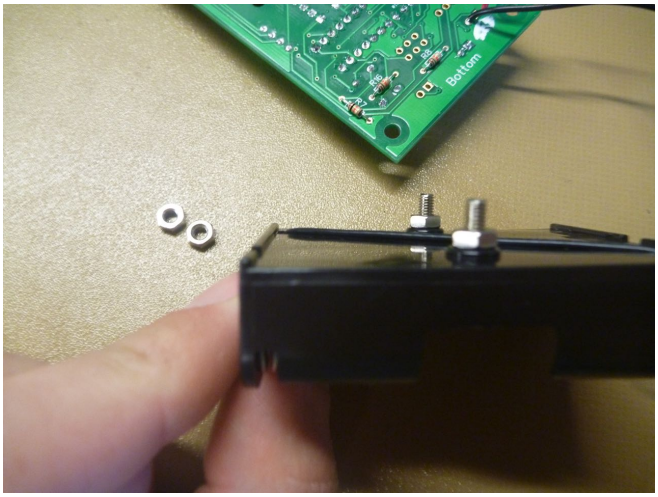
Now that the soldering is over, we just need to screw some nuts and bolts together and you'll be completely done with the assembly. This part is much quicker than the soldering. I feel like this part is best explained with pictures, so I'll put a few in here for bolting down the battery pack.



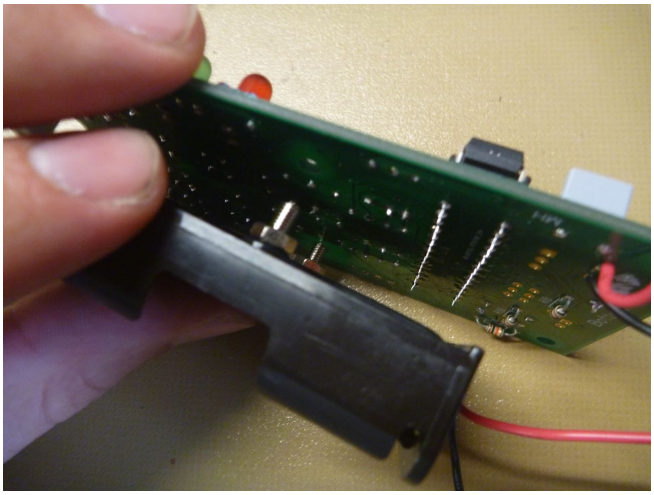
Battery Pack Assembly 1: We need 4x M3 hex nuts and 2x6mm M3 countersunk bolts



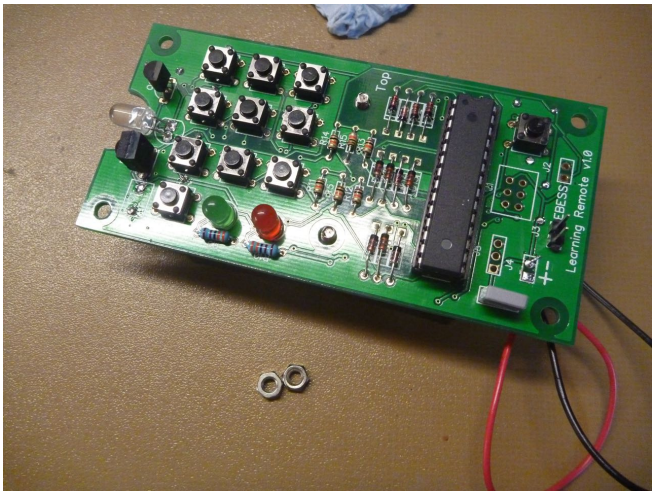
Battery Pack Assembly 2: Put the two countersunk bolts into the battery pack



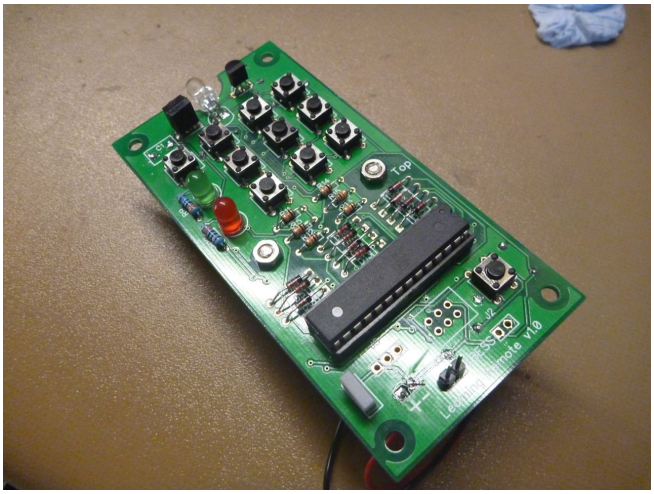
Battery Pack Assembly 2: Screw two of the hex nuts tightly onto the bolts coming out of the battery case



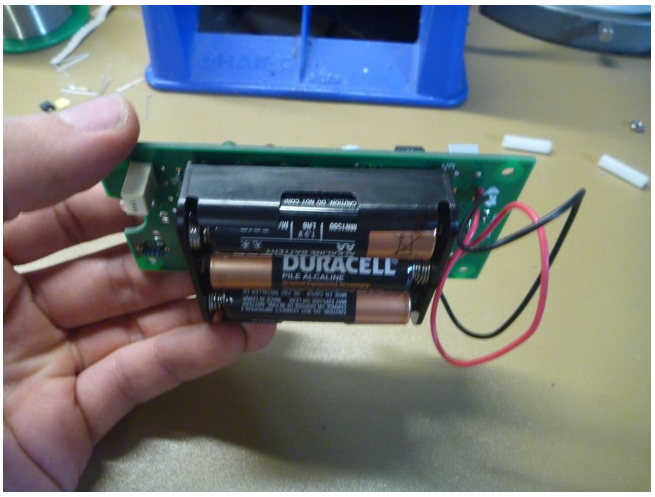
Battery Pack Assembly 4: Put the battery pack bolts through the two middle holes on the bottom side of the PCB



Battery Pack Assembly 5: Almost there, we just need the last two hex nuts

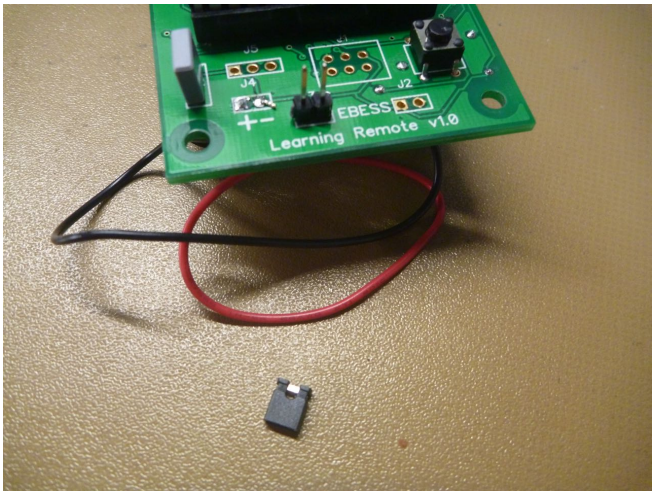


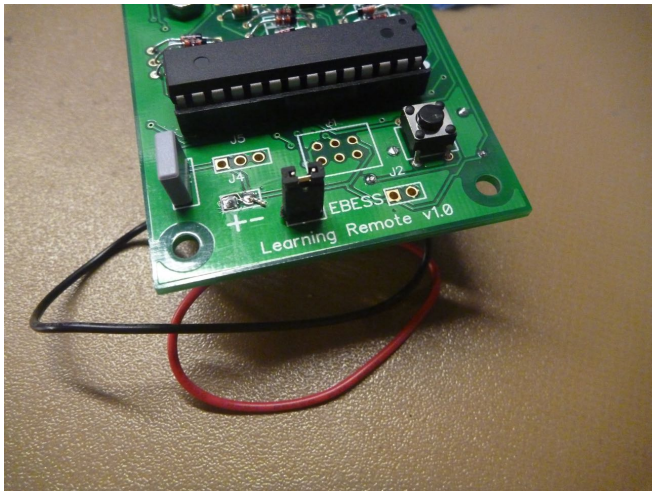
Battery Pack Assembly 6: Now just screw the last two hex nuts on the protruding bolts and you'll be done



Battery Pack Assembly 7: Put three AA batteries in (note orientation of batteries in case)

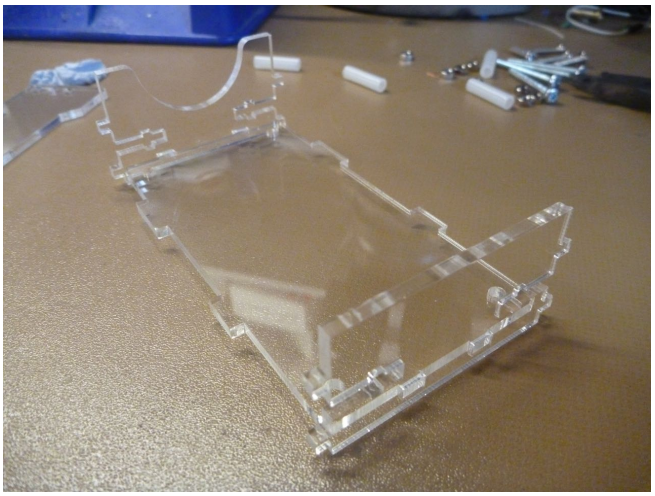
Now we just have to put your AA batteries in (not included), put the jumper shunt in, and your remote should be fully functional and ready to record your signals.



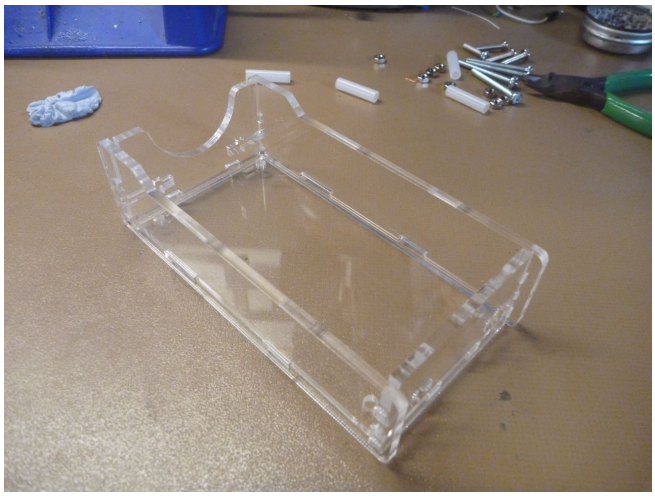


Battery Pack Assembly 8: Put the jumper shunt on, and your remote should be all powered up!

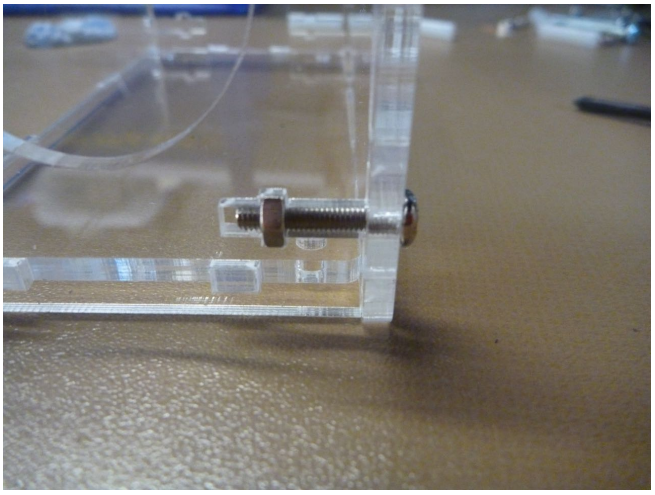
Let's assemble the case now. Grab your five pieces of acrylic, eight M3 hex nuts, four 30mm M3 bolts, four 16mm M3 bolts, and four nylon spacers. The pieces can only go on the correct way, so if the holes don't line up, it's the wrong piece!



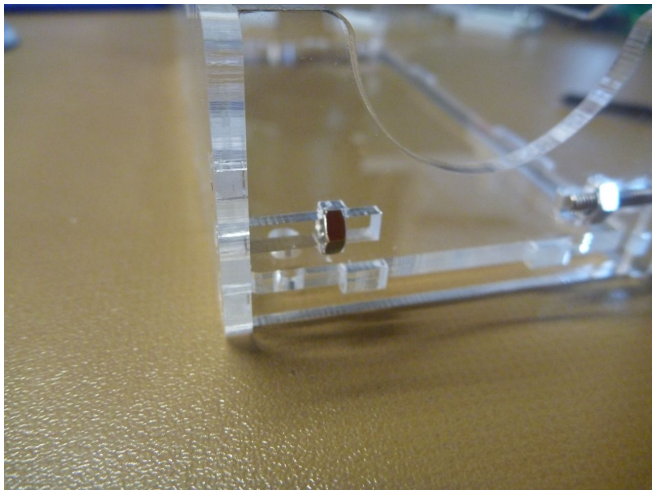
Case Assembly 1: Put the front and rear pieces on the base



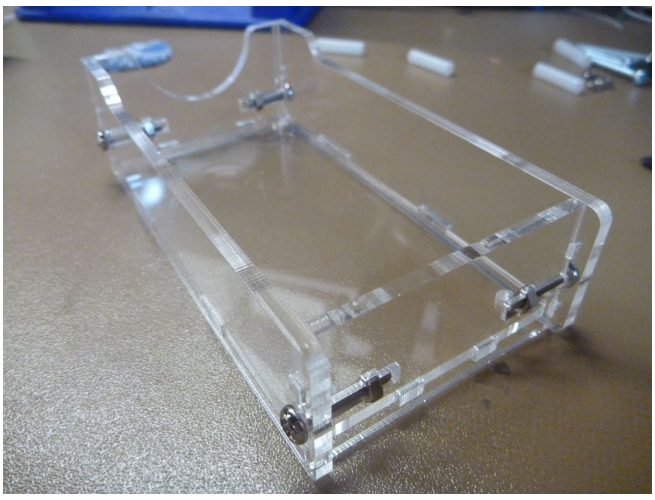
Case Assembly 2: Put both sides on



Case Assembly 3: Securing the case



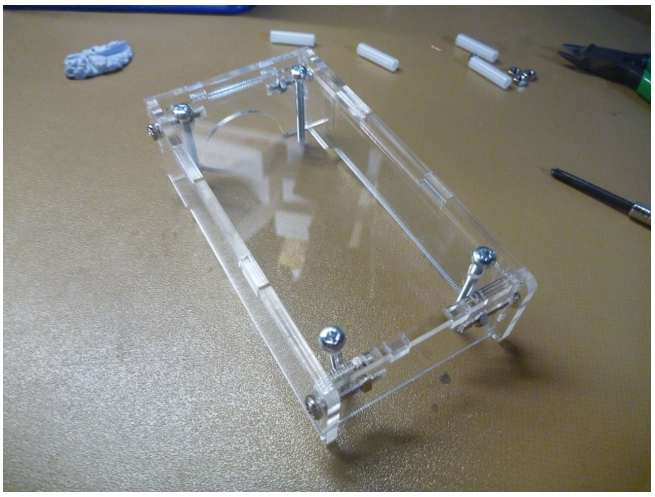
Case Assembly 4: Hex nut positioning



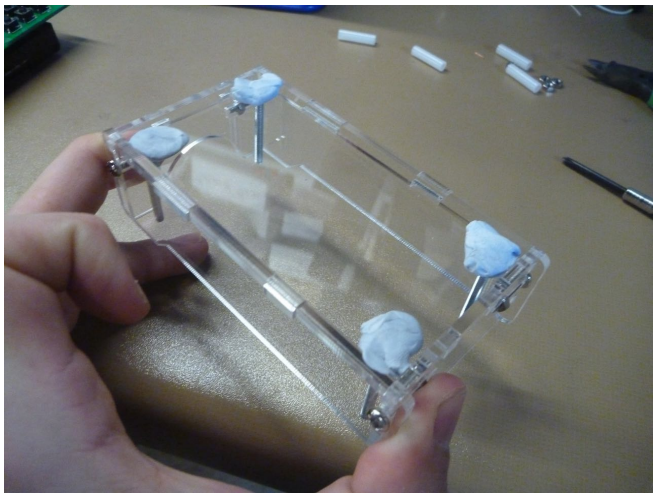
Case Assembly 5: Case secured!

Now we're going to secure the case. A hex nut will fit snugly in any of the given slots as shown above, and then you just need to insert an M3 16mm bolt and screw it in. The acrylic will hold the nut in place while you're screwing the bolt. Screw all four 16mm bolts and nuts into the slots and your case should be secure.

All that's left in this process is to secure the PCB, so let's get that done. As shown below, we turn the case upside down, and insert all four 30mm M3 bolts in the four holes in the base plate.

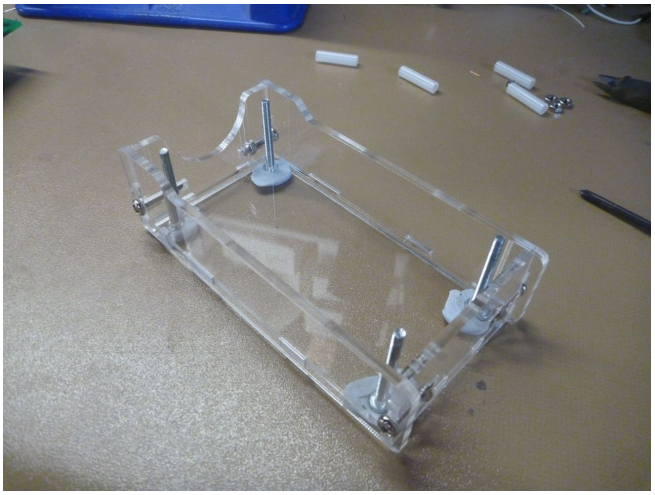


Case Assembly 6: Turn the case upside down and put all four 30mm M3 bolts in the holes

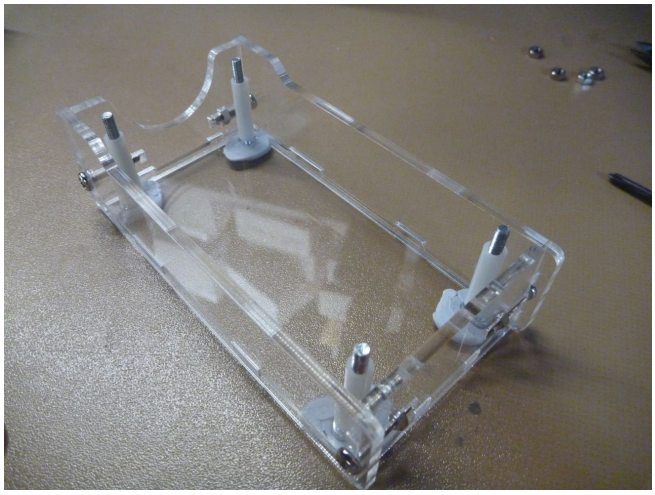


Case Assembly 7: Hold them in place with something like blu-tack

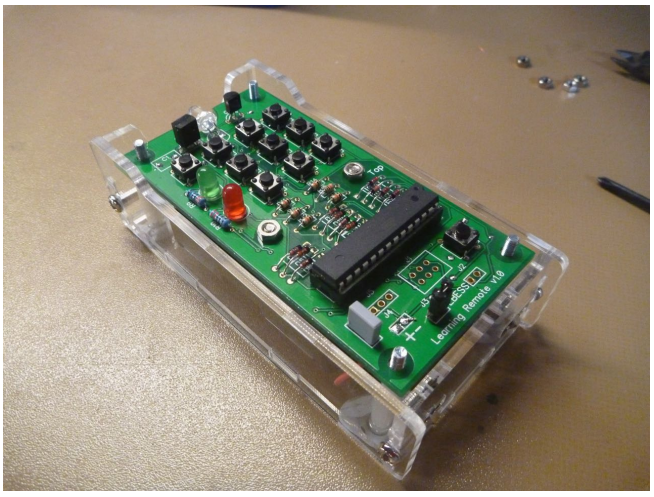
Now turn the case the right way up again, and slide the nylon spacers on each bolt. Then put the PCB on the bolts, noting orientation in the pictures. Then secure the hex nuts and we're all done with the assembly, congratulations on all your hard work.



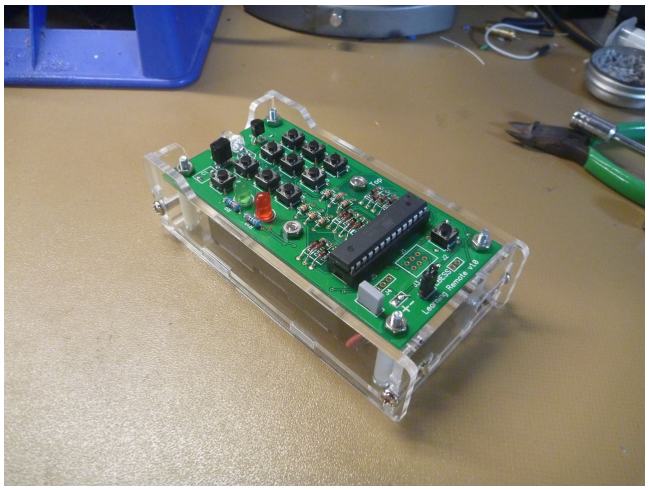
Case Assembly 8: Turn the case the right way up



Case Assembly 9: Slide the nylon spacers on

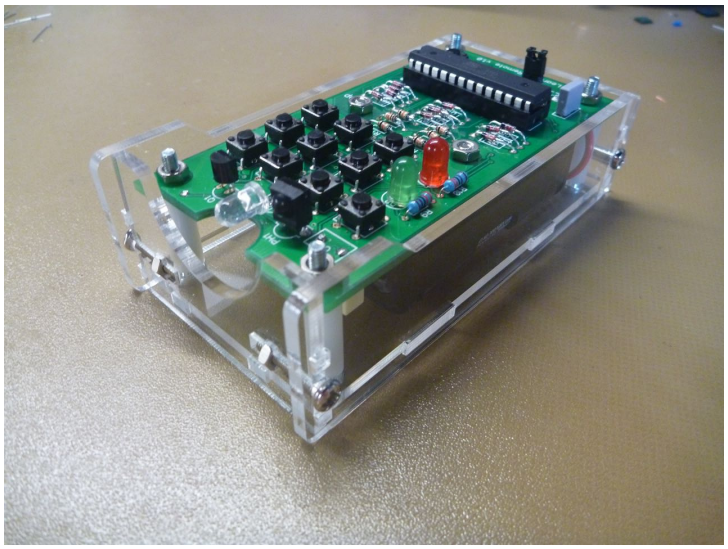


Case Assembly 10: Place the PCB on the bolts



Case Assembly 11: Put the nuts on the bolts and tighten them up (not too tight)

Remove the blu-tack on the bottom, and that's it!



Operation

Turning the device on and off

Once batteries are in the battery pack, and everything is correctly soldered and connected, the remote control will be on and awaiting your button press. Turning the remote off is simply a matter of removing the batteries.

Operating modes

There are two modes of operation for this device, transmission and recording. They are explained below, and the transition between modes is explained in the 'toggle button' section further down.

1. Transmission mode

   The remote will default to this when turned on. It will go straight into a power saving mode until a button is pressed. If any of the command buttons are pressed, the remote will transmit the signal associated with that button (if it has a signal associated with it), and then it will immediately go back to sleep to conserve power.

2. Recording mode

   In this mode, the user will be able to record signals and save them to buttons. Upon entering this mode, the red LED will light up, indicating that we are in recording mode.

   The user then must press a command button to choose which button they want the signal to be saved to. When a command button is pressed, the green LED will light up, indicating that it is ready and waiting to receive the user's signal. At this point, the desired signal just has to be transmitted into the IR receiver at the top left of the remote.

   Having the two remotes as close as possible guarantees the best possible recording. Once the IR receiver receives the signal and processes it, the two LEDs will turn off on the remote, and the device will revert to transmission mode.

   If the lights do not turn off, that means that the remote either did not register the signal (move your remote closer, and make sure there is no obstruction), or it could not record your signal. If this occurs, try exiting and re-entering recording mode and trying again, with your remote closer.

   If at any point the user wants to exit recording mode, they must simply press the toggle button. Additionally, the recording mode will automatically exit if it has not received a signal after 20 seconds, going straight back to transmission mode.

Example of the recording procedure

User enters recording mode, with red LED on. User then presses a command button to select which button they want the signal to be saved to. Upon pressing a command button, the green LED turns on. The user then points their air-conditioning/TV/DVD/whatever remote at the lens of the IR receiver at the front left corner of their remote, and presses and releases a button on their chosen remote to play a signal. Almost immediately after pressing and releasing the chosen remote's button, the learning remote's LEDs should turn off, and it will go into transmission mode. The user can now press the command button again to transmit the signal that they just recorded.

Toggle button

At the bottom right of your remote, you have a lone button. This is used to swap between transmission and recording mode. When in transmission mode, pressing the button will turn on the red LED, indicating that you have moved into recording mode. In recording mode, pressing the button will turn off all LEDs, indicating that you are in transmission mode.

Programming the firmware

Your microcontroller should come pre-programmed with the necessary code. The source code and hex file will be available online if you would like to reprogram it at any point, or if you would simply like to look at the code.

Tips on recording your own signals from different devices

This section will discuss various things I've come across while developing the remote.

- The power on/power off signal appears to be the same signal for most devices (TVs, etc), so you only need to record it once. The key exception that I've found is air-conditioning, which (for the system I've used) requires a unique on and off signal, so you will need reserve two buttons for those.
- 1-2cm is a good range to be from the receiver when recording signals.
- The remote most likely won't be able to record commands from universal remotes, as they spit out multiple commands in a row. This also means that excessively long signals won't be able to be recorded.
- If you are having difficulty recording signals, try doing it in an area with lower lighting and/or different lights. Different ambient light could be saturating the sensor and messing up the recording process
- If you have a buggy signal, try recording it again, perhaps trying a different recording environment
- Incandescent lights may make for a difficult recording environment, as they can churn out infra-red light, which can confuse the remote's sensor

Theory

You don't have to worry too much about this section. It's only for those who are curious about learning about some new concepts and seeing what's behind the remote. Since there are a few sections to the theory, this will be broken up into different parts. This is where all the really cool stuff happens.

Binary amplitude shift keying (modulation)

If this interests you, consider taking COMS4105 down the road.

Brief introduction

You've heard of AM before, amplitude modulation, be it in the context of radio, TV, or some other system. Amplitude modulation works by having two signals: one called the 'carrier wave', and the 'information signal'. The carrier wave carries the information at its fixed frequency (hence the name). This frequency is the really important part here, which could be anything like 693kHz for 4KQ radio, or around 37kHz for TV remotes (important clarification: The 693kHz refers to the frequency of the electromagnetic wave, which is the carrier wave for 4KQ, but the 37kHz refers carrier wave frequency of the TV remote; it is not the frequency of the IR electromagnetic wave, which is ~320THz).

At the transmitting side, the information signal is basically multiplied onto the carrier wave, and then the receiver listens at the carrier's frequency, filters the signal, and extracts the information out of it. AM gets its name from the fact that you modulate (control, vary) the amplitude of a carrier signal.That's a pretty brief overview of AM, which is a method of analogue modulation, but here we're going to be looking at digital modulation, which is used in almost everything around you: Wi-fi, Bluetooth, mobile phones, Go-Card scanners,payWave, student ID cards, TV remotes, and much more.

We discussed AM above because now we're going to look at its digital counterpart: ASK (amplitude shift keying). The same concept applies from AM in that we have a carrier wave at a fixed frequency, but now we are modulating in a digital fashion, instead of analogue. This simply means that we have discrete amplitude levels to represent digital symbols.

These 'digital symbols' are numbers, being 0, 1, 2, 3, 4, etc (or in binary: 000, 001, 010, 011, 100). The number of symbols in your communication system depends on how many you can handle and how many you want. For example, in your learning remote control, only two symbols are used: 0 and 1, making it binary amplitude shift keying (BASK). You could have 8ASK, meaning that the symbols 0 to 7 are used, requiring you to be able to detect 8 unique amplitudes.

Modulation and demodulation

BASK (OOK, PAM) is easy to modulate and demodulate because we can represent a 0 as the lack of the signal being present, and a 1 as the signal being present. This is demonstrated in Figure 17 below. At the transmitter, we don't transmit anything when we want to send a 0, and we transmit at the carrier frequency

when we want to send a 1. Figure 18 shows an oscilloscope screenshot of a real remote control transmission. You'll notice a time delay between the modulated and demodulated signals, which is due to the fact that there is inherent delay in the demodulation process, but also that I pointed the transmitting remote (pink signal) away from the receiver, which meant that the signal had to spend more time in the air, reflecting off surfaces before it eventually reached its destination.

In this particular system, we are waiting for a signal to be in the same state (present or not present) for a certain amount of time before we can confidently determine that it was a 1 or a 0 that was meant to be transmitted.The numbers on the bottom of the various plots below represent the bits that were gleaned from the signal after being demodulated. In this ideal case in Figure 17, it is very easy to see that the presence of a signal demodulates to a 1, but we'll investigate some cases below where it's a bit more difficult.

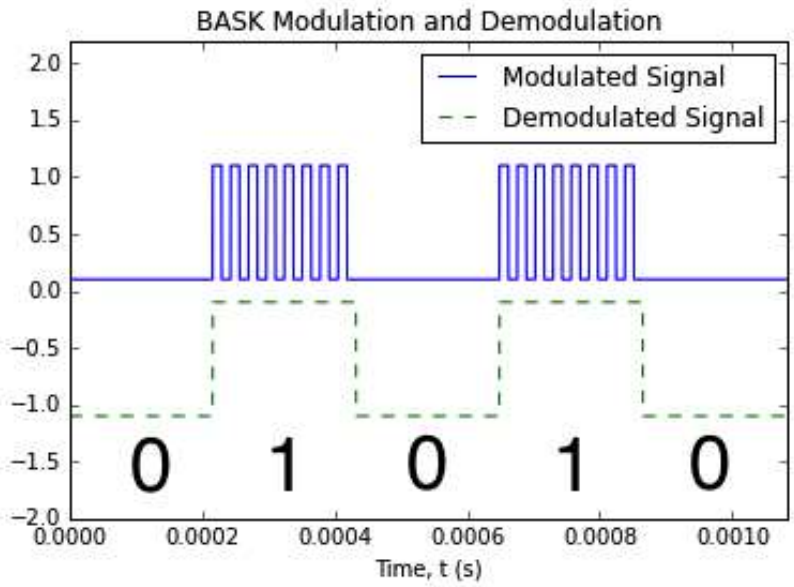


Figure 17: Ideal BASK Modulation and Demodulation

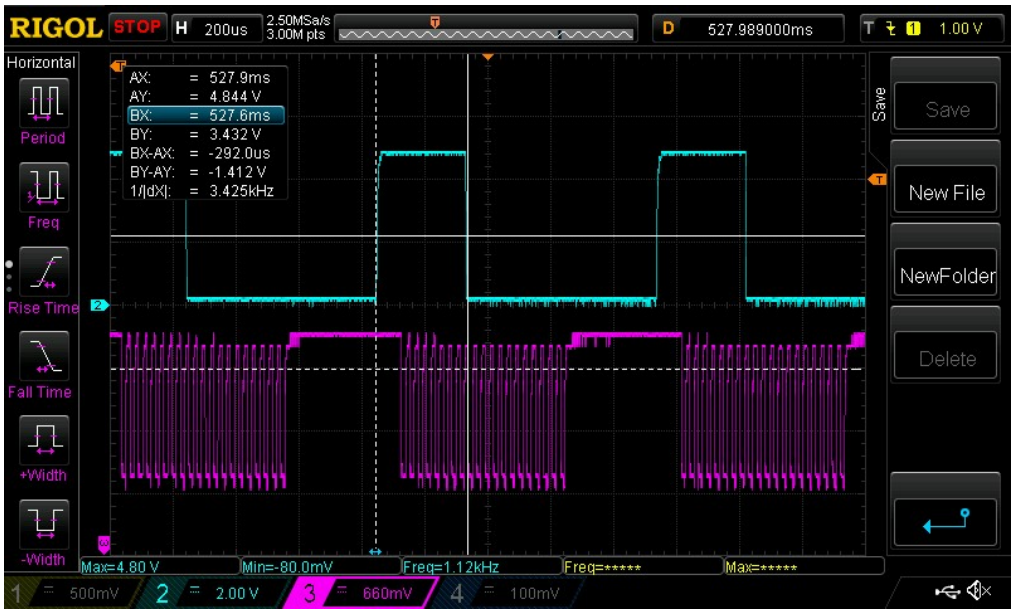


Figure 18: Real life remote controldemodulation and modulation

Why modulate?

So you might consider looking at Figure 17, 'Why do we even need to modulate? Why can we not simply transmit a high pulse for the required time for a 1? Is there any need to use a carrier wave?' In some situations, you can certainly get away with very simple communication systems, but modulation offers us some robustness to noise and interference, and it additionally allows selectivity if we want to have multiple communication systems running at different frequencies simultaneously.

Figure 19 shows how modulation can help us resist the influence of noise. Let's keep talking about the remote control, and say that there's already a bunch of IR flashing around your room. Your remote control wants to transmit a 0, and you might detect that by setting a threshold. If the amplitude is above a certain level, you say 1, and if it's below the threshold, you say 0.

However, as we have mentioned, what if there's already some IR light reaching the receiver? How does the receiver know what it should listen to and what it should ignore? A simple threshold can work, but there could be large spikes of noise that could trick your receiver into thinking a different bit has been transmitted. To combat this, your receiver looks for a precise frequency of a signal, which in our case is ~37kHz, and it ignores everything else. As a result of the desired signalhavinganoticeably larger amplitude than the noise, this can be achieved easily by filtering the signal at the carrier frequency of 37kHz. This yields the clean demodulated signal.

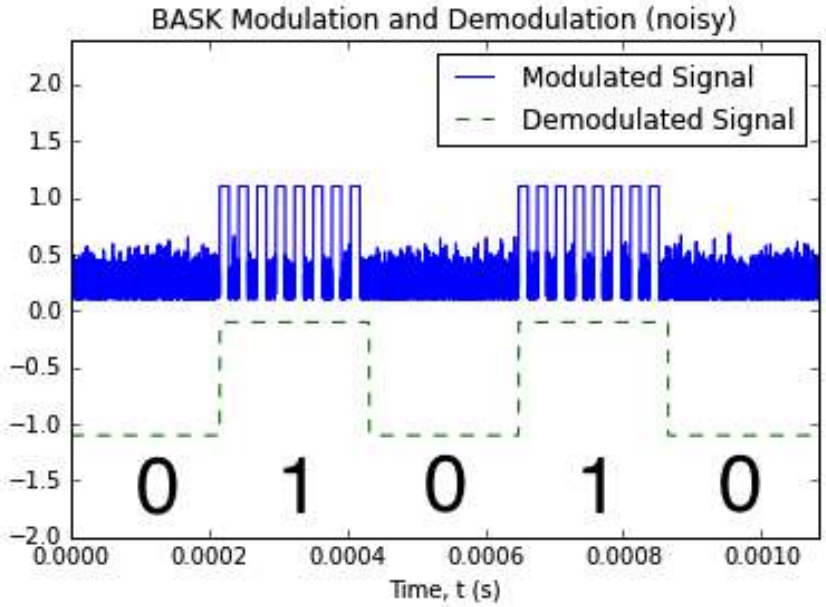


Figure19: BASK Signal with Additive White Gaussian Noise (AWGN)

Now let us consider the case of not only noise, but also other systems communicating at the same time. Figure 20 shows that by locking onto our carrier frequency and extracting only the information at that frequency, we can ignore others that are transmitting around us.

You can observe that there are two unwanted signals present. There is one that is wider (meaning that it has a lower frequency), and a signal that is thinner (meaning that it has a higher frequency). Thanks to our demodulator only listening at our carrier frequency, it was able to ignore other transmissions that were occurring at the same time. Our data still comes out correctly at the other end.

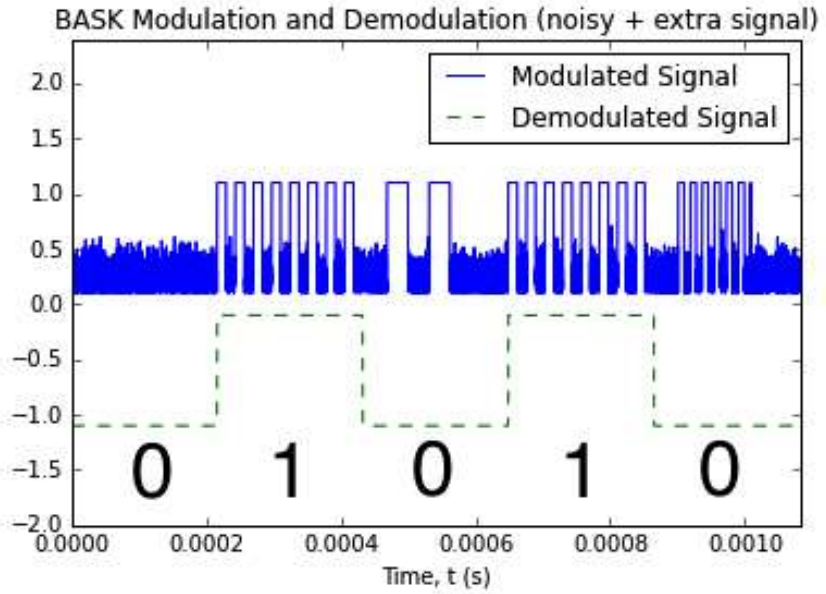


Figure 20: BASK Signal with AWGN and Interfering Signals

However, there comes a point in every communication system when we simply cannot see what's being transmitted anymore. Interference or noise could be too great that our demodulation and detection algorithms just give out nonsense. I really doubt that this would occur often with a TV remote, unless you're outside in the and you have a bit of bad luck.

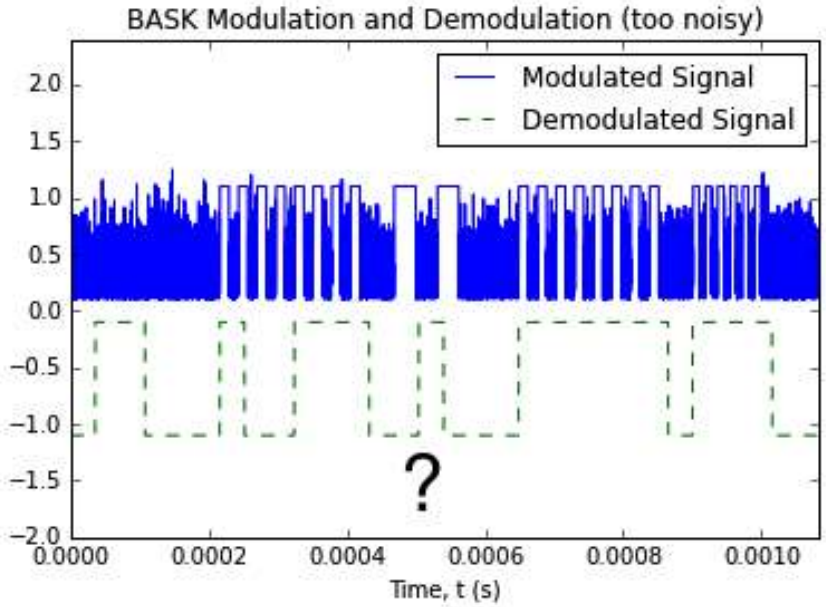


Figure 21: BASK Signal with Significant Noise and Interfering Signals

While we have outlined BASK here, most remote controls don't exactly use it, but recognising BASK demodulation helps capture other signals. Some may use pulse distance modulation (PDM), pulse width modulation (PWM), Manchester modulation, and the list goes on. As a result of us wanting our remote to be able to handle these different modulation schemes, we record the time that each pulse is high, and the time that each pulse is low. This allows us to re-implement whatever modulation we received, so long as it only keys two amplitude levels.

PDM works by varying the null time between each pulse to differentiate between a 0 and a 1. The pulse part usually lasts the same amount of time, and the null time has the different lengths to represent data, as shown below

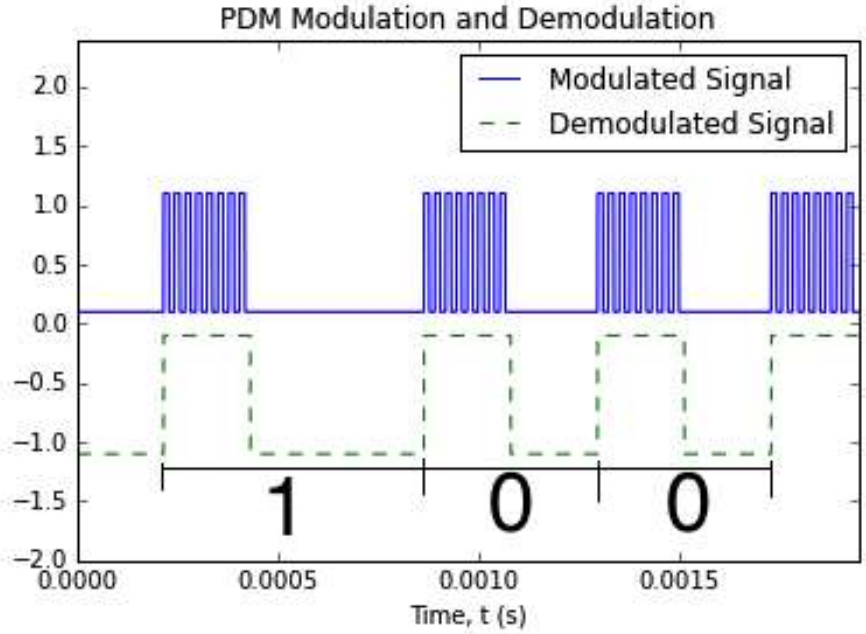


Figure 22: Pulse Distance Modulation (example only, 0s and 1s not always like this)

You may already have experience with PWM, which works by having a fixed length of time for each bit; it just varies how long of this period the pulse takes up. A short pulse could be a 0, and a longer pulse could be a 1, as seen below.

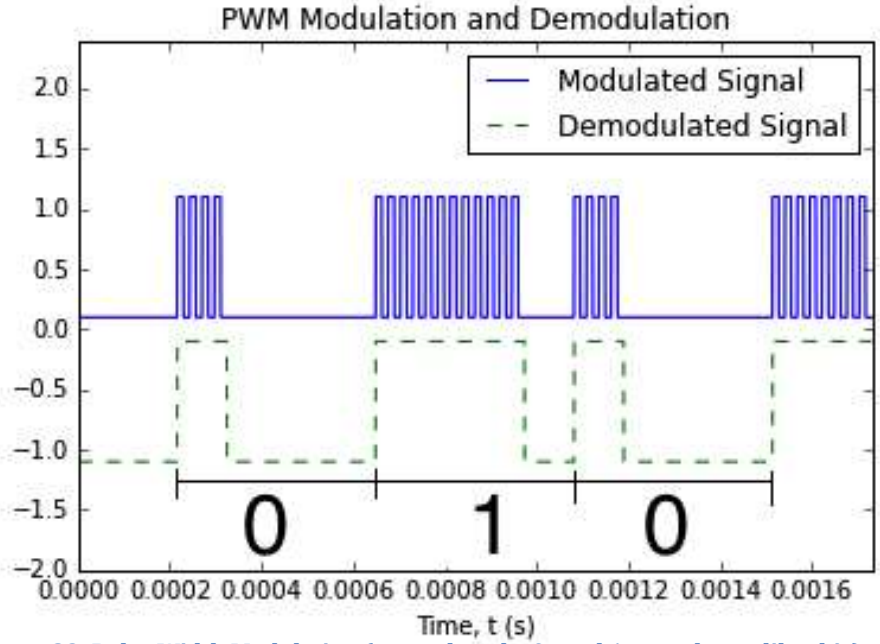


Figure 23: Pulse Width Modulation (example only, 0s and 1s not always like this)

Huffman coding (data compression)

This content is introduced in CSSE3010 and COMS4105.

Explanation

Huffman coding is a data compression algorithm that is easy to understand. It works well in a visual manner, so we'll discuss some things briefly, and then dive into examples.This is used to compress the recorded IR signals on the remote, in order to make them small enough to store in the EEPROM.

This data compression algorithm works by determining how many unique symbols there are to encode in a dataset, and how probable each symbol is to occur. The more probable a symbol is to occur, the less data it uses to represent it. A more probable symbol will occur more often, so it is advantageous to represent common things with less data, as it will consume less data overall. If that didn't make sense, here's an example.

We have a dataset that we want to transmit: {A, B, A, A, A, C, A, B, D, A, B, A}. We can see that the symbol A occurs the most, and is therefore the most probable. Now we count them all up.
A: 7 times
B: 3 times
C: 1 time
D: 1 time

So we have a total of 4 unique symbols being transmitted a total of 12 times. A naïve approach would be to simply give each simple a binary representation and send them, with the receiver having a reference library to get the symbol back out. In binary we can do the following

A = 00 (code length = 2)
B = 01 (code length = 2)
C = 10 (code length = 2)
D = 11 (code length = 2)

Each symbol has equal binary length of 2, so we know that the number of bits required to transmit this sequence is 12×2=24 bits.We have an average code length of 2. Let's see if we can do a bit better and apply the Huffman coding algorithm.

A has a probability of 7/12 (0.5833) of occurring, B has 3/12 chance (0.25), and C and D each have 1/12 chance (0.0833) of occurring. With this information, we are now going to construct a Huffman tree. We start off with each unique symbol being a root of the tree, and we add the two lowest probabilities together, and form a branch. If multiple probabilities are the same, just pick a route and be consistent. We continue this until all the probabilities add up to 1. The tree is constructed on the next page.

Once the tree is constructed, we need to go along all the branches and assign a 0 and a 1 to each pair. So long as you are consistent and careful with how you

apply this, there will be no issue. My preferred method is to simply start at the top of the tree and give a 0 to the left branch and a 1 to the right. I then travel down each node and repeat. This will guarantee us unique codes (that we need!).

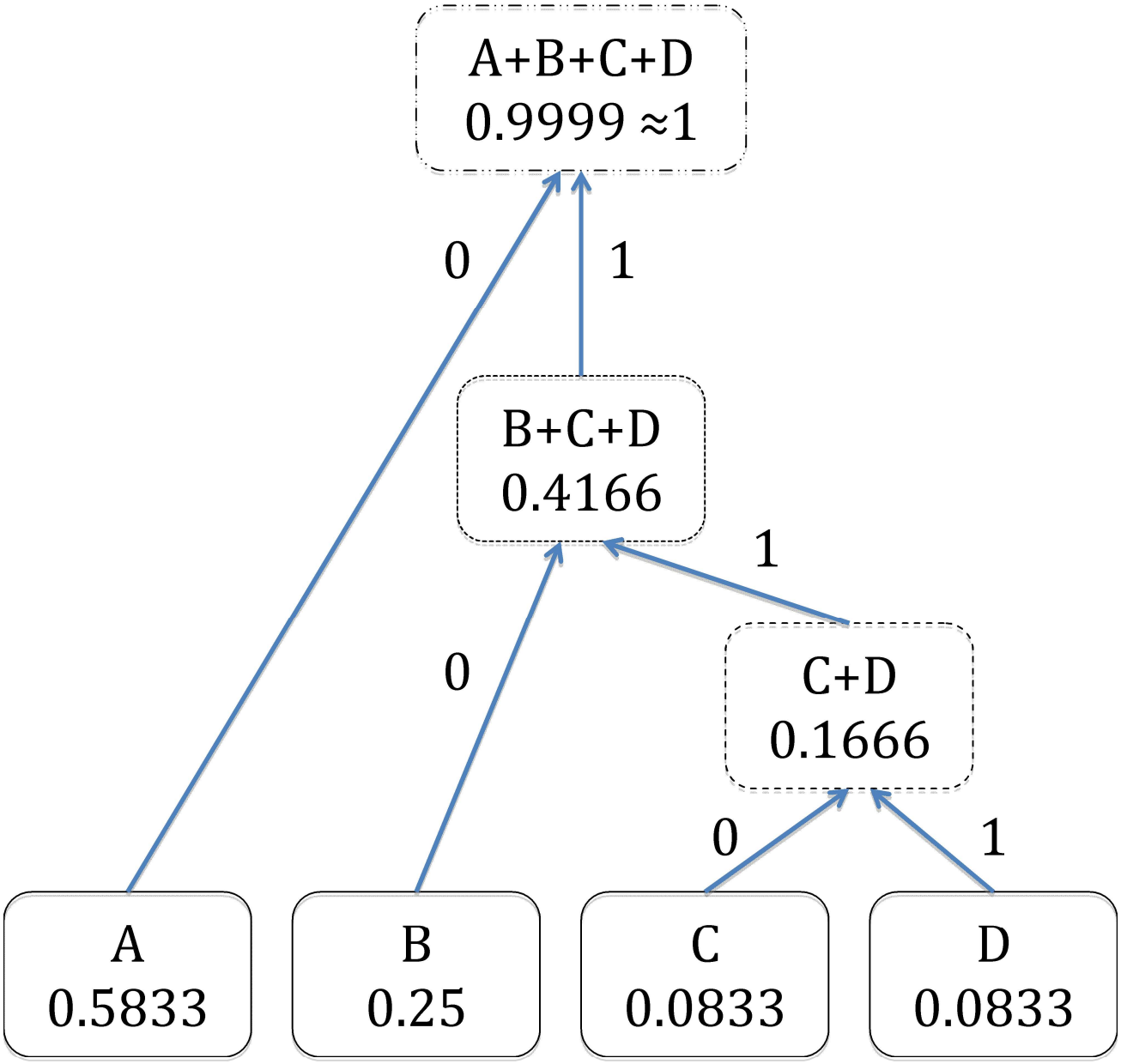


Figure 24: Huffman Tree

Now that our tree is all nicely constructed, we can determine new binary codes for each symbol. We start from the top of the tree, and work our way down to each root, collecting bits as we go. So our new binary codes would be:

A = 0 (code length = 1)
B = 10 (code length = 2)
C = 110 (code length = 3)
D = 111 (code length = 3)

Now let's calculate how many bits we would need to transmit this whole dataset. We do this by multiplying how many bits per symbol by how many of that symbol appears. For example: [code length] × [number of appearances] + … + …

Number of bits $= 1 \times 7 + 2 \times 3 + 3 \times 1 + 3 \times 1 = 19$ bits. So we have successfully compressed our previous 24 bits into just 19 bits, and the average code length has been reduced from 2 to 1.5833 (19/12). This is a compression of 79.2% of the naïve scheme.

Entropy

Now that we have just compressed our data, it is a good time to introduce the concept of entropy. You may have heard of this in the context of thermodynamics before, but now we'll be talking about information theory. Entropy can be described as the maximum that we can compress a dataset. If necessary, Huffman coding can approach the entropy of a dataset with a few iterations. All you need to do is calculate the entropy and then compare it with the average code length.

The relevant formula here is: Entropy = $-\Sigma p log_2(p)$, where p is the probability of a symbol occurring. We go along every symbol in the dataset and apply the formula. First, recall our dataset is {A, B, A, A, A, C, A, B, D, A, B, A}, with probabilities: A (p = 0.5833), B (p = 0.25), C (p = 0.0833), and D (p = 0.0833). Then apply the formula.

$$Entropy = -\Sigma p log_2(p)$$
$$Entropy = -[0.5833 log_2(0.5833) + 0.25 log_2(0.25) + 0.0833 log_2(0.0833) + 0.0833 log_2(0.0833)]$$
$$Entropy = 1.5508$$

Our average code length was 1.5833, which is just slightly above the entropy of the system; so this is an optimal code that performs near the theoretical limit (102.1% of entropy). It is perfectly fine to use this compressed code now.

Chasing entropy

You can stop reading about Huffman coding here if you would like, but continue reading if you're curious about trying to get closer to the entropy of the system.

A disparity can come about if one symbol is considerably more probable than the other symbols (approx. >0.5 probability), so if you are interested, we can in fact get closer to the entropy by increasing the number of symbols, as this will reduce the individual probability of symbols. Hence we will get closer to entropy.

We do this by instead of considering A, B, C and D as the only symbols, welook at the probabilities of multiple symbols as one and thus increase the number of symbols. So here, if we consider two symbols at a time, we will have 16 symbols. Let's construct the table of new symbols on the next page. I will go to one more decimal place to slightly increase accuracy

New Symbol	Probability Calculation	Probability
AA	0.58333×0.58333	0.34027
AB	0.58333×0.25	0.14583
AC	0.58333×0.08333	0.04861
AD	0.58333×0.08333	0.04861
BA	0.25×0.58333	0.14583
BB	0.25×0.25	0.0625
BC	0.25×0.08333	0.02083
BD	0.25×0.08333	0.02083
CA	0.08333×0.58333	0.04861
CB	0.08333×0.25	0.02083
CC	0.08333×0.08333	0.00694
CD	0.08333×0.08333	0.00694
DA	0.08333×0.58333	0.04861
DB	0.08333×0.25	0.02083
DC	0.08333×0.08333	0.00694
DD	0.08333×0.08333	0.00694
Sum		0.99995 ≈1

Table 1: New Symbol Probabilities

Now brace yourself, because we're about to construct a monster Huffman tree on the next page. We'll use the space here to get the average code length and new entropy.

New Symbol	Binary Code	Code Length
AA	00	2
AB	010	3
AC	1000	4
AD	1010	4
BA	011	3
BB	1001	4
BC	11000	5
BD	11001	5
CA	1011	4
CB	11100	5
CC	111100	6
CD	111101	6
DA	1101	4
DB	11101	5
DC	111110	6
DD	111111	6

Table 2:New Symbol Binary Codes

Our dataset can be interpreted as 6 new symbols now: {AB, AA, AC, AB, DA, BA}. Let's recalculate the entropy.

$$Entropy = -\Sigma p log_2(p)$$

$$\begin{aligned} Entropy = -(&0.34027 log_2(0.34027) \\ &+ 0.14583 log_2(0.14583) \\ &+ 0.04861 log_2(0.04861) \\ &+ 0.04861 log_2(0.04861) \\ &+ 0.14583 log_2(0.14583) \\ &+ 0.0625 log_2(0.0625) \\ &+ 0.02083 lo\ _2(0.02083) \\ &+ 0.02083 log_2(0.02083) \\ &+ 0.04861 log_2(0.04861) \\ &+ 0.02083 log_2(0.02083) \\ &+ 0.00694 log_2(0.00694) \\ &+ 0.00694 log_2(0.00694) \\ &+ 0.04861 log_2(0.04861) \\ &+ 0.02083 log_2(0.02083) \\ &+ 0.00694 log_2(0.00694) \\ &+ 0.00694 log_2(0.00694)) \end{aligned}$$

$$Entropy = 3.10202$$

Our total code length is 19, which is the same number of bits as before. The new average code length is 19/6 = 3.16667, which is 102.1% of entropy. We can see that there was no real improvement here, so we should stick with the first go. However, now you know how to get closer to entropy if your data needs it.

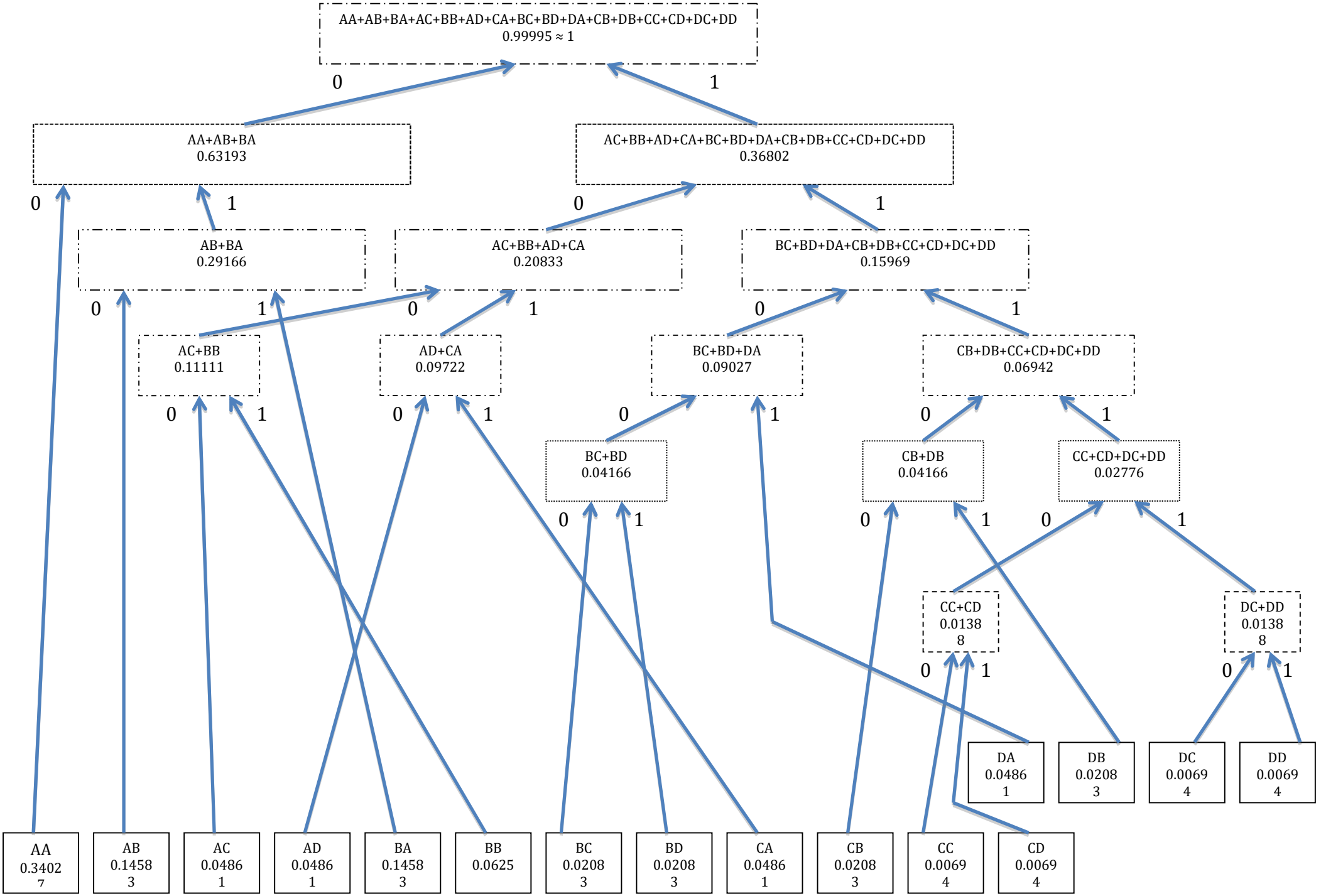


Figure 25:Huffman Tree that should never be done by hand

Circuit Explained

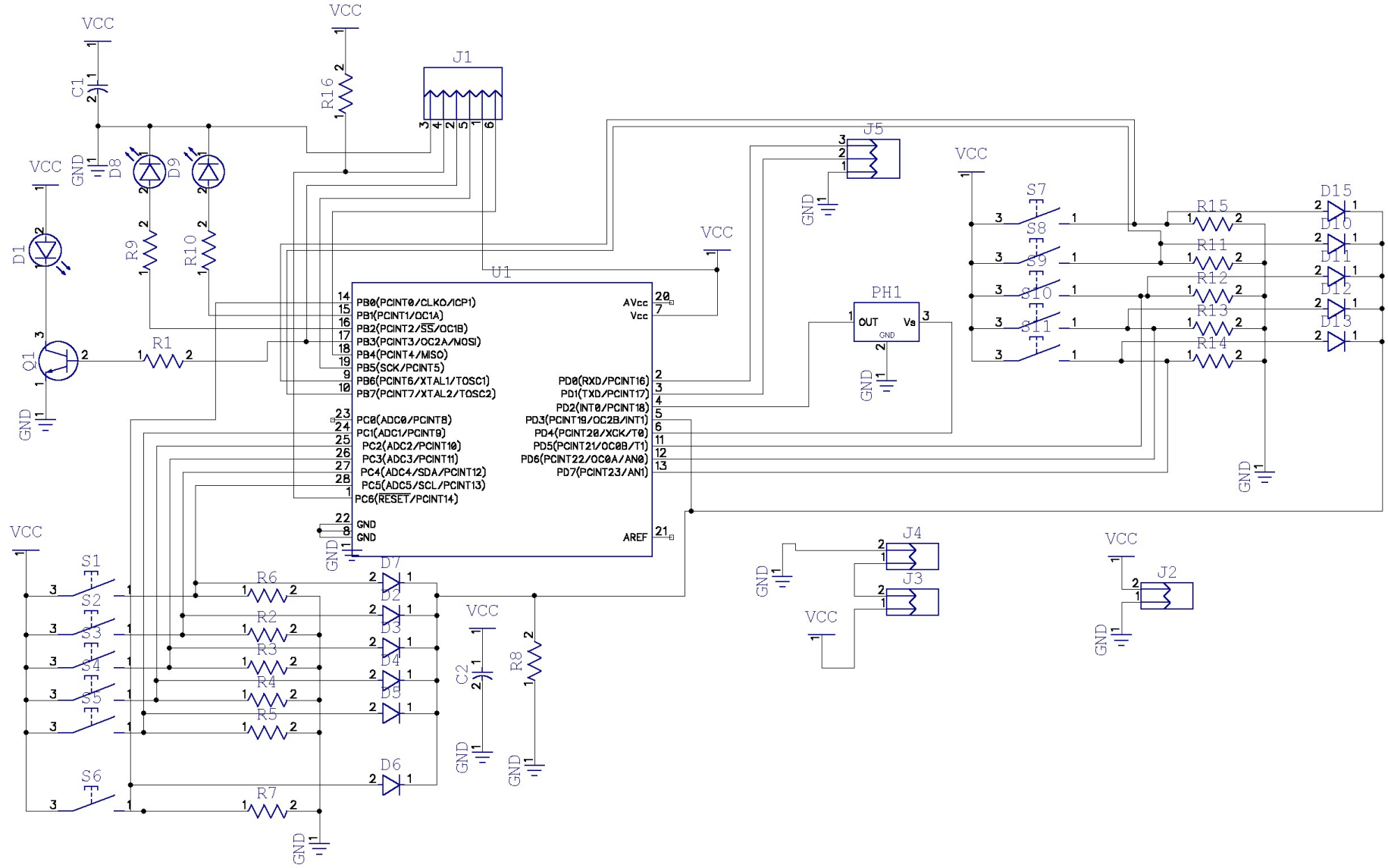


Figure 26: Circuit Schematic

The schematic

It's okay if you don't understand all of the schematic on the previous page; you don't have to.

The big block in the centre is the microcontroller, the Atmega328. Its representation is not equivalent to how everything is laid out in real life; the components are set out for clarity in schematic mode. We discuss some reference designator (the names of components on the schematic) prefixes below, but here you can access a more extensive list:
https://en.wikipedia.org/wiki/Reference_designator

IR emitter and demodulator

Near the left side of the schematic (when the page is oriented in a landscape view), you will see the components with the part designator (or reference designator) D1 and Q1. D is the common prefix for a diode, and those arrows pointing off D1 indicate that this is a light-emitting diode, an LED (the IR emitter). Q is the prefix for transistors, and here we are using an NPN bipolar junction transistor (BJT). Vcc means the positive voltage source from your battery, and GND means ground, where the negative terminal of your battery goes. As you may guess, R is a resistor.

The IR emitter requires more current than the microcontroller can supply from its pin for a bright pulse, so we use the BJT. The terminals 1, 2 and 3 on Q1 correspond to parts called the emitter, base and collector, respectively. The basic idea is that you supply a small amount of current through the base, and then the collector draws an amplified amount (often 100-300x) of that current from Vcc. The sum of those two currents then goes out through the emitter to ground. So with a small current from the microcontroller, we can dump a larger current through the IR emitter to ensure that it can transmit a good distance. The resistor R1 is on the base of the BJT to ensure that the BJT doesn't draw too much current from the microcontroller, which would then be multiplied through the IR emitter, and could blow it up.

Just to the right of the centre of the schematic, you will notice a unit called PH1. This is the IR demodulator (TSOP4836). The part designator is an arbitrary one given in the model of the component provided by Vishay (the manufacturer). As a result of this component requiring a very low current (~800µA), it is possible to power it directly off the microcontroller. It usually is not advisable to power integrated circuits like this off the microcontroller pins, but I did it in this case only because it drew such little current, and it was necessary for some tricky power saving that will be explained in the microcontroller section of this document.

The OUT pin of PH1 is where we get the demodulated signal described in the theory section. The microcontroller can directly read this and then do its operations.

Debugging

Scattered around the place, you will notice boxes of varying sizes with arrows inside them. They will have the part designator prefix J. This means that they are jacks (or jumpers as well in this case). This means that on the PCB, they are pins that stick out of the board that allow us to interface with it. These are mostly useful for development and debugging, and offer no real purpose to the end-user.

J3 and J4 are used for current measurement. We do this by connecting the + and – of the battery pack into J4 as per usual, but instead of having the jumper shunt on the pins of J3, we place a multimeter across the two pins (in series) and put it in current measuring mode. This means that all the current for the circuit has to flow through the multimeter to get from one terminal of the battery to the other, and thus the multimeter can tell us the current consumption of the circuit. This is useful for observing power saving modes and calculating battery life.

J2 is used for voltage measurement. We can check the voltage level of the battery pack by simply measuring the voltage across these two pins.

J5 is used for serial communication to a computer. This basically means that we can get the microcontroller to send messages and data to a computer, while it's running, so that we can monitor how the program is going in real time. This is probably the most useful debugging feature on the PCB, as it allows the microcontroller to report things that are going on, allowing for rapid debugging.

Finally, J1 is for programming the microcontroller. All of your microcontrollers should come pre-programmed, but in case something goes wrong, this will be used. An Atmel (company that makes the Atmega328) In-System Programmer (ISP) will be attached to the 6 pins that then interface with the microcontroller.These jumpers are all very useful debugging tools that will help you solve problems in your own projects.

The rest

Now that we've addressed the more complicated parts of the circuit, it's time to finish off everything else. We'll start with the buttons, which are called tactile switches. Theswitches are denoted by the S prefixed designators. All of them are connected to their own microcontroller pin so that they can be checked. There are methods where you can matrix buttons or multiplex them that would consume less pins, but there was not much to do on this circuit, so I opted for a pin per button.

You'll notice that each button has a resistor (10kΩ) connected to its output and then to ground. This is called a pull-down resistor. Pull-down resistors ensure that if there is no voltage at a node, it will be pulled down to ground (0V). This is important here because nodes that are not connected to a voltage (being connected to a microcontroller pin doesn't count) are in a 'floating' state. That is to say that its voltage level is floating and is determined by electromagnetic radiation in the air, human fingers touching nodes, and other things that we have no control over. We care about the node voltage of the buttons in particular because we know that a button has been pressed when a high voltage (Vcc) is

observed on the microcontroller pin that is attached to the button. When the button is not pressed, we want it to be 0V, which is why we have a pull-down resistor.

However, if we didn't have the pull-down resistor, the pin associated with the button would be floating. This can produce false positives whereby the microcontroller thinks that a button has been pressed, and then it will act on that. So we have the pull-down resistors there to make sure that the buttons are always at the potential that we want, whether it be pressed or un-pressed.

Now you may have noticed that there are also a bunch of diodes connected from all of the buttons to a single pin on the microcontroller (still with a pull-down resistor on that line). This is will be explained more in the microcontroller section, but the purpose of this is basically to allow all the buttons to be able to wake up the remote when it goes to sleep to conserve power. In the lowest power saving mode, there are only two pins that can wake the microcontroller up, so we connected all the buttons to one of those pins so that it can wake up and perform its task. All the buttons are connected through diodes because this means that they won't be able to trigger false positives on other buttons. The diodes separate the voltages by only allowing current to flow in the pointing direction of the diode, and not the other way around (unless in extreme circumstances).

While we're on the topic of pull-down resistors, you'll notice that near J1, there is a resistor connected to Vcc. This is called a pull-up resistor, which has the opposite effect of the pull-down resistor: a node disconnected from a voltage will be pulled up to whatever voltage Vcc is. This pull-up resistor is on the reset pin of the microcontroller, which is quite important. If the reset pin is pulled low (0V), the microcontroller will reset its code and start again. We don't want this to occur sporadically throughout use of the device, so we ensure that it stays high.

Getting towards the end, you'll notice D8 and D9 over on the left hand side, which are in fact visible LEDs, not IR emitters. These are the two indicator LEDs that allow the user to know ifthey're in receiving mode, and whether the device is ready to receive a signal or not.

Last, but absolutely not least, we have the two capacitors, C1 and C2. These are between the Vcc power rail and GND, and are called decoupling capacitors. Batteries and power supplies typically cannot react instantaneously to rapid changes in power demands, whereas capacitors can. The IR emitter draws comparatively large amounts of power very quickly when it's transmitting, so we place a capacitor right as close as possible to the BJT and IR emitter. When the increased power demand begins, the capacitor supplies what it can, trying to maintain a stable voltage level. After a short period of time, the power supply will be able to start supplying the needed current, at which point the capacitor can start recharging to do the job all over again. Often you will find a few decoupling capacitors of a few nanofarads and a microfarad or so. This is because capacitors act as short circuits at higher frequencies and can shunt high frequency variations in voltage straight to ground to stop circuit interference.

The Microcontroller

What's a microcontroller?

You will learn a great deal about microcontrollers in CSSE2010, CSSE3010, and CSSE4011.

A microcontroller is basically a very small computer. It has a processor, memory, and pins that can interface with other electronics. You can program it to do a whole host of things, as has been done in this project.

Microcontrollers are used in embedded systems, which is a circuit designed for a specific purpose. You will have embedded systems in your phone, car, space shuttle, etc. Basically microcontrollers are everywhere, and once you learn how to use them, you can create whatever you like! There are plenty of resources on the internet to get you started, and CSSE2010 is particularly great for an introduction.

This project

In this remote control, we are using a microcontroller called the Atmega328, which is made by Atmel. In microcontrollers, you have an architecture, which defines how they operate at the machine language level. The Atmega328 uses something called AVR architecture. ARM is another architecture that you will hear around the place that is often in more advanced microcontrollers.

As with any electronics, the datasheet is your best friend. It will tell you precisely how to use the microcontroller of your choice, and you should always read it carefully. You will have the best time if you can avoid making assumptions as much as possible, and always check the datasheet.

You will find yourself typically programming your microcontrollers in embedded C, which is C with a few extras. Throughout programming your microcontroller, you'll find yourself using the IO (input/output) pins, which can be used to read a voltage level as an input, or you can output various voltage levels. These are the key to interacting with components and actually getting tasks done.

Of course, there are some other tools on microcontrollers that can help you do more. On the Atmega328, you have 3 timers, which the remote control uses to time how long pulses on recorded signals last, to modulate a transmitted signal, and to perform some internal functions, like timing out if you've been in the receive mode for too long.

There's also a special and very important timer called the Watchdog timer, which appears in almost all microcontrollers. It is a timer that, if enabled, is always working in the background, counting down. A primary purpose of it is to prevent system hang-ups. If something goes wrong in your code, and your project crashes, the Watchdog will count down to 0 and reset your device. The way to prevent this system reset is to constantly refresh the Watchdog timer throughout

your code. So if it doesn't reach a refresh point in time, something will have gone wrong in your code, and thus a system reset could help save the day.

Another special piece of hardware inside the Atmega328 that we take advantage of is the EEPROM (electrically erasable programmable read only memory). It is special memory that is persistent (non-volatile), which means that if you unplug the batteries from your device and plug it back in later, it will remember whatever you store there. This is where we save the recorded IR signals. So if you need to change the batteries at any point in your remote, you won't have to re-record all of your signals.

There is another type of persistent is flash, which is where all of your program code is written to when you program the device. While it is possible for the microcontroller to write to its own flash memory while it is running, I hadn't used EEPROM before (it's pretty easy to use), so I opted to use that in this project. Self-writing to flash can be a bit risky if you don't know what you're doing, as you could overwrite the program code, causing unpredictable behaviour. EEPROM is completely removed from this, and is designed to be accessed easily.

Contact & Credits

I have questions

There's a lot of text in this document, but I haven't even been able to come close to comprehensively covering all the content (for microcontrollers alone, that's basically impossible). However, if you have any nagging questions, or you're just plain curious, please feel free to contact me about anything even barely related to this project, and I'll try my best to answer.

All materials for this project, like the code, this guide, the PCB Gerber files, and the laser-cut case files can be found here on the EBESS website.

Who made this?

Marcus Herbert, the EBESS Technical Officer of 2016. If you would like to see some of his other projects, his website is below.
Email: marcusherbert7@gmail.com
Website: marcusherbert.net

Thanks to my brother, Jeremy, who gave me suggestions along the way, and let me bounce ideas off him.He operated the laser cutter so that you could all have a case for your PCB. Thanks to ITEE for access to the laser cutter.

Also thanks to Andrew, Michael, Philip and Luca, who tested the final kits and gave helpful feedback. Jeremy, Vai, Ellie and Cam were of great help in sorting the kits.

Final thanks to all the EBESS execs, who have been supporting this idea and who want to help students out.

Your 2017 EBESS student executives wish you all the best for the year to come. Hopefully we'll be able to help you along the way!

Further Reading

1. Huffman Coding
2. Modulation
3. Demodulation
4. Amplitude Shift Keying
5. Atmega328 Datasheet
6. Diptrace (PCB and Schematic software)
7. Pulse Position Modulation